A formal study of Free Software distributions

Jaap Boender

Université Paris Diderot — Paris 7
École doctorale de Sciences Mathématiques de Paris Centre

THÈSE
pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS DIDEROT
Spécialité : **Informatique**

présentée par
Jacob Pieter BOENDER

Directeur : Roberto DI COSMO

Étude formelle des distributions de logiciel libre

A formal study of Free Software Distributions

soutenue le 24 mars 2011
devant le jury composé de :

M. Roberto DI COSMO	directeur
M. Jesús M. GONZÁLEZ-BARAHONA	examinateur
M. Carsten SINZ	rapporteur
M. Diomidis SPINELLIS	examinateur
M. Jean-Bernard STEFANI	examinateur
M. Ralf TREINEN	examinateur
M. Peter VAN ROY	rapporteur

Contents C

Résumé R

De quoi s'agit-il ?
— FERDINAND FOCH

Comme l'indique le titre, dans cette thèse, il sera beaucoup question des distributions du logiciel libre (aussi connu sous l'abréviation F/OSS, *free and open source software*).

Ces distributions sont extrêmement hétérogènes. Elles contiennent des logiciels de différentes provenances ; écrits dans des langages différents, avec des calendriers de publication différents et avec des procédures différentes.

Pour gérer cette hétérogénéité, et pour avoir une façon simple et unique d'installer des logiciels, les systèmes de *paquetage* ont été développés. Ceux-ci consistent en l'emballage d'un logiciel dans un paquet, qui contient des données supplémentaires utilisées par un logiciel approprié, un *gestionnaire de paquets*. Le gestionnaire sert à installer les paquets et le logiciel qu'il contient de façon presque automatique.

Les systèmes de paquetage diffèrent selon les distributions, mais les principes sont communs : une distribution a plusieurs dépôts, dont chacun contient plusieurs paquets, reliés entre eux par des relations spécifiques, notamment les *dépendances* et les *conflits*. Une dépendance d'un paquet a un autre indique que le premier paquet ne peut pas être installé sans que l'autre soit installé aussi ; un conflit entre deux paquets indique que ces deux paquets ne pourront jamais être installés en même temps.

Les dépendances peuvent être *disjonctives*, c'est à dire qu'un paquet peut spécifier une dépendance sur plusieurs paquets, dont au moins un doit être installé pour satisfaire à la dépendance.

Tout ceci fait que le problème de l'installabilité d'un paquet est d'une complexité comparable au problème SAT. Quand on y ajoute le fait que les distributions d'aujourd'hui ont une taille importante (la version la plus récente de Debian contient 22 000 paquets), il devient clair qu'il est très important d'avoir des algorithmes rapides et efficaces.

Les quatre sujets principaux abordés dans cette thèse se résument comme suit :

- D'abord, nous présentons un modèle formel qui réunit les propriétés principales des systèmes de paquetage les plus courantes, et nous identifions des relations sémantiques entre paquets qui peuvent être utilisées pour trouver des erreurs et assurer la qualité des distributions de logiciel libre ;

- Ensuite, nous présentons des algorithmes efficaces pour manipuler des dépôts de paquets et calculer les relations mentionnées ci-dessus ; tous ces algorithmes ont été implémentés dans la langage de programmation OCaml, et incorporés dans une librairie de manipulation et analyse de paquets qui s'appelle dose3 ;

- Nous avons encodé notre modèle dans l'assistant de preuves Coq, et utilisé cet encodage pour vérifier quelques-uns des théorèmes les plus importants qui correspondent aux étapes les plus compliquées des algorithmes déjà présentés ;

- Finalement, nous avons validé nos algorithmes sur des distributions de logiciel libre existantes, et nous présentons une analyse extensive de la structure générale de ces distributions, notamment les caractéristiques dites «petit monde» de la structure du graphe sous-jacent.

La théorie des paquets

Dans cette partie de la thèse, qui correspond aux chapitres 2 et 3, nous commençons par expliquer les concepts communs entre les différents systèmes de paquetage.

Systèmes de paquetage

(a) Dépendances et conflits (b) Paquet virtuel
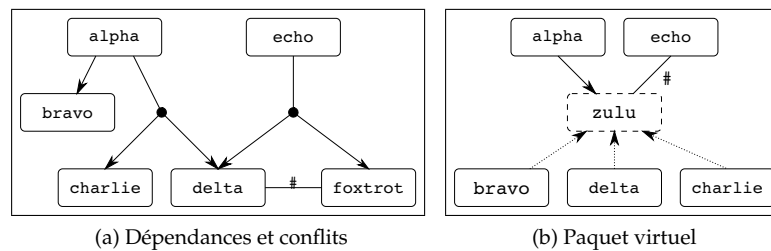
FIGURE 1 – Dépôts d'exemple

Considérons la figure 1a. Ici, le paquet alpha a une dépendance simple sur le paquet bravo, ce qui exprime la nécessité d'installer bravo dès lors qu'on veut installer alpha. En revanche, l'autre dépendance d'alpha est disjonctive sur charlie et delta. Alors pour installer alpha il faudrait aussi installer soit charlie, soit delta, soit les deux.

La situation est semblable pour echo : pour installer echo, on a besoin de delta ou foxtrot. Cependant, il n'est pas possible ici d'installer les deux en même temps, parce que delta et foxtrot sont en conflit (indiqué par le dièse).

Dans la plupart des systèmes de paquetage, il existe la notion de *paquets virtuels*. Comme leur nom l'indique, ce sont des paquets qui n'existent pas vraiment, mais qui peuvent être *fournis* par des autres paquets. Considérons la figure 1b : ici, le paquet zulu est un paquet virtuel, qui est fourni par bravo, charlie et delta.

Dans ce cas, alpha a une dépendance sur zulu ; une dépendance sur un paquet virtuel peut être satisfaite par n'importe quel fournisseur du paquet virtuel, et la situation revient donc à une dépendance disjonctive d'alpha sur bravo, charlie ou delta.

Quant à `echo`, il y a un conflit entre ce paquet et `zulu`. Un conflit avec un paquet virtuel se traduit par un conflit avec chaque fournisseur de ce paquet virtuel : autrement dit, `echo` est en conflit avec à la fois `bravo`, `charlie` et `delta`.

Il y a des différences de détail entre les systèmes de paquetage (dont les plus utilisés sont le système de Debian et RPM), mais ils sont tous conformes au schéma expliqué ci-dessus.

Relations sémantiques entre paquets

Les relations entre paquets qu'on a détaillées dans la section précédente ne nous donnent pas toute l'histoire. Par exemple, le simple fait qu'il y a une dépendance entre deux paquets ne signifie pas forcément que l'un doit être installé pour installer l'autre (la dépendance pourrait être disjonctive). Il pourrait aussi y avoir des paquets qui, sans être en conflit direct, ne sont néanmoins pas installable en même temps[1].

Pour pallier à ce problème, nous installons des relations *sémantiques* qui nous aident à voir plus clairement la structure d'un dépôt.

Commençons par les dépendances. Nous définissons qu'il existe une *dépendance forte* entre un paquet p et un autre paquet q (tous les deux contenu dans un dépôt R) si et seulement si :

- p est installable[2] dans R ;

- chaque installation de p dans R contient q.

Avec cette relation de dépendance forte, on résume l'essentiel de la dépendance : un paquet a des dépendances fortes sur tous les paquets qui lui sont absolument indispensables. Notons aussi que la relation de dépendance forte, contrairement aux dépendances normales, est transitive.

Il est également intéressant de considérer les dépendances fortes en sens inverse : un paquet dont beaucoup d'autres paquets dépendent fortement est nécessairement un paquet important dans la distribution : si ce paquet avait un défaut, ceci pourrait avoir un impact sur beaucoup d'autres paquets.

Néanmoins, de par la transitivité des dépendances fortes, il est possible qu'un paquet ait beaucoup de «prédécesseurs forts» sans pour autant être très important lui-même. Cette situation est illustré par la figure 3.3c sur la page 40. On y peut voir que `quebec` a beaucoup de prédécesseurs forts. Le fait que `quebec` dépend fortement de `romeo`, et la transitivité des dépendances fortes, font que tous les prédécesseurs forts de `quebec` sont aussi des prédécesseurs forts de `romeo`. Ainsi, `romeo` paraît comme un paquet plus important que `quebec`, même si le nombre de ses prédécesseurs est en quelque sorte *expliqué* par la dépendance forte entre `quebec` et `romeo`.

Pour éviter cette situation, on introduit les *dominateurs*, un concept qui est connu dans le domaine des graphes de contrôle de flux : un paquet p domine un autre paquet q si et seulement si tous les chemins de dépendances fortes qui mènent vers q passent par p[3].

---

[1] Une exemple de cette situation est la figure 3.5 sur la page 46.

[2] Nous ajoutons ce point pour éviter les dépendances fortes triviales ; sinon, un paquet non-installable aurait des dépendances fortes sur chaque autre paquet dans la distribution.

[3] La définition donnée dans la thèse est différente, mais nécessite un peu plus d'explications. On démontre toutefois dans la thèse que la définition donné ici est équivalente à celle donné dans le chapitre 3

En utilisant les dominateurs, on peut nettoyer la structure des dépendances fortes de façon que les paquets qui paraissent importants, mais ne le sont pas vraiment, comme expliqué ci-dessus, soient enlevés.

Ce qu'on peut faire pour les dépendances, on peut faire pour les conflits : deux paquets p et q d'un dépôt R sont en *conflit fort* si et seulement si on peut installer p et q séparément dans R, mais non pas ensemble.

Ici encore, on résume l'essentiel de la relation de conflit : deux paquets qui ne peuvent pas être installés en même temps ; pour avoir cette situation, il n'est point nécessaire qu'il y ait un conflit direct entre les deux paquets. Par contre, nous avons démontré que pour qu'un conflit fort existe entre deux paquets, il doit exister un chemin de dépendances (normales) entre chacun des deux paquets et un conflit (théorème 3.25).

Algorithmes et outils

Dans la section précédente, nous avons noté que le problème de déterminer si un paquet est installable dans un dépôt est de complexité égale au problème SAT, c'est à dire NP-complet.

On a également vu que les relations sémantiques dépendent aussi de l'installabilité des paquets. Vu la taille des distributions, il n'est pas envisageable de simplement calculer la totalité des relations en vérifiant l'existence d'une relation pour chaque paire de paquets.

Dans le chapitre 4, nous proposons donc des algorithmes plus efficaces, dont le fonctionnement repose sur des théorèmes présentés dans le chapitre 3.

Pour les dépendances fortes, l'algorithme proposé utilise d'abord le fait que pour qu'une dépendance forte existe entre deux paquets p et q, q doit être présent dans n'importe lequel installation de p ; pour trouver toutes les dépendances fortes de p, on peut donc se borner à contrôler tous les membres d'un ensemble d'installation de p quelconque. En plus, nous utilisons le fait qu'une dépendance conjonctive est automatiquement une dépendance forte (corollaires 3.7 et 3.2).

Le calcul efficace des conflits forts repose essentiellement sur le théorème 3.25. Puisque, pour avoir un conflit fort entre deux paquets p et q, il est nécessaire qu'il existe un chemin de dépendances (normales) de p et q jusqu'à deux paquets qui sont en conflit, on peut rassembler tous les conflits forts en commençant par les conflits directs (dont il y a relativement peu), et en remontant les dépendances en sens inverse. Ainsi, on obtient tous les paires de paquets qui pourraient être en conflit fort, ce qui réduit l'espace de recherche de façon importante.

Pour les dominateurs, on utilise le théorème 3.19 qui démontre que notre notion des dominateurs est équivalent à celle utilisée dans le domaine des graphes de contrôle de flux. On peut ensuite utiliser l'algorithme de Tarjan [LT79] pour calculer rapidement le graphe des dominateurs.

Dans le chapitre 5, nous présentons les outils qui ont été crées en faisant usage de ces algorithmes. Notamment, il s'agit de `dose`, qui a été conçu comme une librairie de manipulation et analyse de distributions, et qui inclut tous les algorithmes présentés ci-dessus.

Formalisation

Comme on a vu dans la section précédente, les algorithmes présentés ont été optimisés en utilisant des théorèmes qui permettent de réduire l'espace de recherche.

Pour être sûr qu'on ne réduit pas trop l'espace de recherche, il faut s'assurer de la validité des théorèmes utilisés. Pour ce faire, nous avons utilisé l'assistant de preuves `Coq` pour formaliser une partie de la théorie des paquets, et notamment pour vérifier les théorèmes les plus importants utilisés dans nos algorithmes.

Une explication des méthodes utilisées dans cette formalisation est le sujet du chapitre 6.

Validation et analyse

Dans les chapitres 7 et 8, nous présentons des résultats d'expériences faites avec les outils présentés précédemment.

D'abord, nous parlons des temps d'exécution des algorithmes. Théoriquement, ce sont toujours des algorithmes de complexité NP-complet ou assimilé, mais en utilisant les optimisations mentionnées, on peut obtenir des réductions assez importantes qui permettent de calculer les graphes de relations sémantiques dans un temps raisonnable ; ainsi, il devient possible de faire un calcul quotidien.

Ensuite, nous nos intéressons à la structure de la distribution. Dans des publications précédentes ([LW04] et [NNR09], il a déjà été démontré que les distributions de logiciel libre ont un graphe sous-jacent qui présente le phénomène du petit monde ; nous affirmons que c'est le cas en bien précisant notre méthodologie, ce qui n'a pas été le cas dans les publications citées.

Le phénomène du petit monde est surtout intéressant pour les conclusions qu'on peut en tirer sur la structure de la distribution. Un graphe petit monde est un graphe qui a des chemins relativement court entre ses noeuds ; on peut aussi diviser les noeuds d'un graphe petit monde dans deux catégories : les noeuds a forte connectivité (il y en a peu), et les noeuds a faible connectivité (il y en a beaucoup).

Les graphes de distribution de logiciel libre ont la particularité d'être dirigée ; on peut alors distinguer trois types de noeuds distincts :

- Des noeuds avec beaucoup d'arêtes sortantes, mais peu d'arêtes rentrantes ; ces noeuds présentent des paquets de haut niveau, appelés les *meta-paquets*, qui sont utilisées pour installer facilement une famille de logiciels, comme KDE ou GNOME ;

- Des noeuds avec beaucoup d'arêtes rentrantes, mais peu d'arêtes sortantes ; ces noeuds présentent des paquets de bas niveau, des libraires notamment, comme par exemple la librairie standard de C qui est nécessaire pour une grande partie des autres paquets ;

- Des noeuds avec peu d'arêtes, rentrantes ou sortants.

Dans le chapitre 8, il y a plus d'informations qui confirment l'existence de cette structure.

Contributions et perspectives

Dans cette thèse, nous avons présenté une modélisation formelle des distributions de logiciel libre, avec des méthodes pour améliorer la gestion de qualité qui utilise cette modélisation. Ces méthodes ont été implémentées en utilisant le langage OCaml, et partiellement vérifiées. Finalement, nous avons utilisé les outils implémentés pour faire des analyses sur des distributions existantes, notamment Debian et Mandriva.

Nous avons présenté des cas réels des erreurs qui ont pu être détectées en utilisant nos méthodes ; leur application quotidienne est rendue possible par les optimisations que nous avons ajoutés. Ceci peut beaucoup aider les éditeurs de distributions à éviter des erreurs, par exemple des paquets non installables.

En continuant ce travail, notamment en complétant la vérification des algorithmes, et en les intégrant dans un langage spécifique, on peut aboutir sur une suite complète d'outils pour la manipulation et l'analyse des distributions. Ceci pourrait aider à garder la qualité des distributions de logiciel libre, même si dans le futur ils continueront à grandir et devenir plus complexes.

Introduction *1*

It has long been a standard method in computing science to divide complex systems into *components* [Szy02].

The basic reason for this is that smaller programs are easier for a human to comprehend, and therefore to design, write, verify, test and maintain. This makes for a quicker design phase, less time spent in implementation, and fewer bugs.

Components can also be reused, especially the more generic ones. This again saves time, because components do not have to be implemented twice. It also results in fewer errors, for the same reason. It is even possible to re-use components developed elsewhere, for example by a third party (for example by using COTS, commercial off-the-shelf products).

Unfortunately, component-based systems have their disadvantages as well, notably in maintenance and evolution. The foremost problem here is the fact that components, as their name already indicates, do not stand alone: they interact with each other.

This interaction brings forth the *relations* between components: these can either be positive (i.e. a component needs another component to function; this is commonly called a *dependency* relation) or negative (i.e. a component can *not* function together with another component; this is commonly called a *conflict* relation).

A component-based system, by its nature, is not stable: components are added, removed and upgraded as a matter of routine. These changes, however, can easily break relationships between components, thus corrupting the state of the entire system or even rendering it unusable.

One specific instance of component-based systems that has become more and more widely adopted over the last two decades are the operating systems based on Free and Open Source Software (F/OSS).

These systems are extremely heterogeneous: every part of the system is developed by a different group; components therefore are implemented in different programming languages, use different release cycles, and have different modalities for downloading and installing. It is easy to see that this can give rise to many compatibility problems.

1.1 F/OSS Software Distributions

In order to at least partially solve these problems, F/OSS operating systems are assembled into *distributions*. This is especially true for the operating systems based on the Linux kernel: there is a great number of distributions based

on this kernel, each with its own specificities. However, other F/OSS operating systems, such as the different varieties of BSD, or OpenSolaris, are also provided in distribution form.

The idea of a distribution is that it creates a coherent system out of the large amount of available software: it allows for a single method of downloading and installing software, and there is one person (the *maintainer*) responsible for integrating the software into the distribution and updating the distribution if a new release is published.

1.1.1 Components as packages

In a distribution, every piece of software becomes a *package*. This package contains the software itself, plus some extra data (known as *metadata*) that describes the package, its dependencies and all the specifics needed to install it on a user's machine. In this way, the user can simply install the package, without having to worry about how exactly to do this: all the necessary information is contained in the package.

The tool used to install a package (using the metadata) is called a *package manager*. The package manager takes care of downloading the package, verifying contents, installing eventual dependencies, making sure there are no conflicts and all actions that are needed to correctly install the software contained in the package (creating specific user accounts, for example).

The packaging format and package manager used vary widely between distributions; in the Linux world, RPM (the RedHat Package Manager) is used by many systems (Fedora, Mandriva, and SUSE, to name a few); another well-known system is Debian's package manager APT with its package format (used by Debian and Ubuntu, amongst others).

1.1.2 The challenge of scale

One of the most important problems that plagues F/OSS distributions today is one of scale. The latest stable version of the Debian distribution (5.0.6, released in October 2010) contains 22 000 packages; the latest version of Mandriva (2010.1) has over 7 500 packages.

Unfortunately, the tools used on both the user side and the distribution side have not notably changed since the first appearance of distributions, now some two decades ago: the SUSE distribution has recently integrated a SAT solver in its package manager, for dependency checking, but most distributions do not yet use even this basic technology.

On the distribution side, the situation is comparable: there are some tools that aid distribution editors in their tasks (one example is Debian's `britney`, which takes care of integration of new version of packages), but these are slow and not formally proven.

1.2 Contributions

The main contributions of this work can be summarised in four broad areas:

- We present a formal framework that captures the essential features of the most common package models, and identify some new semantic relationships among packages that are relevant for finding errors and maintaining quality in F/OSS distributions;

- We present efficient algorithms for manipulating package repositories and compute the relationships mentioned above; all of these algorithms have been implemented in the OCaml programming language, and incorporated in a generic package manipulation and analysis library called dose3;

- We encode our formal framework in the Coq proof assistant, and use it to mechanically verify some of the key lemmas corresponding to the most complicated steps in the aforementioned algorithms;

- We have validated our algorithms on real-world F/OSS distributions, and have performed an extensive analysis of the general structure of these distributions, notably the small world characteristics of the underlying graph structure.

1.2.1 Theory of packages

The details of how packages are represented and manipulated differ quite significantly from one Free Software distribution to the other, but when choosing the right level of abstraction, one can find a remarkably simple and elegant common model that is able to accommodate all the metadata which is relevant for maintaining the quality of a repository. This model has already been presented in [MBDC+06] and is reproduced here with some extensions.

Subsequently, we extend this model with some new semantic relationships between packages. These relationships, in highlighting specific properties of packages, help distribution editors in quickly spotting potential errors and in finding means to correct these errors.

The simplest example is the broken package, a package that cannot be installed under any circumstances. By not only providing a list of such packages, but also an explanation of why they cannot be installed, we help distribution engineers in correcting such packages.

In the same vein, packages that are installable but that, when installed, render a large subset of the distribution non-installable, also are a potential source of errors. Again, since we provide an explanation, distribution editors can easily isolate the source of the problem, and correct it if necessary.

Another way to spot potential trouble is to identify packages that are in some way important to the distribution—for example, because they are depended on by many other packages. We offer a way to identify such packages, so that distribution editors know which packages need extra care and testing in case of changes to the distribution.

Some of the material presented in this part has already been published in [MBDC+06], [ADCBZ09] and [DCB10]; this thesis presents this previous material in its general context, and contains some new additions besides.

1.2.2   Algorithms and tools

The problem of determining whether a package is installable is NP-complete, as already shown in [MBDC+06].

Since the computation of all of the semantic relationships mentioned in the previous section depends ultimately on this installability problem, a naive implementation that checks the existence of a relationship for every pair of packages will take a very long time.

In this thesis, we present algorithms that use the various properties of the packages and their relationships to avoid any superfluous computations. In this way, it becomes feasible to compute the relationships with every major change in the distribution, so that errors can be found as early as possible.

Furthermore, we discuss the implementation of our algorithms as a part of a general distribution manipulation and analysis framework.

1.2.3   Formalisation

For our algorithms, we use several lemmas that allow us to skip a lot of SAT computations. Needless to say, it is very important that these computations can indeed be skipped; in other words, that the lemmas are actually valid.

In order to assure ourselves of this, we have formalised the theory of packages as provided in the previous parts using the Coq proof assistant, and we have formally proven several of the lemmas presented.

1.2.4   Validation and analysis

In this part, we present some practical results obtained by applying our algorithms to some common F/OSS distributions.

To start with, we show that the run time of the algorithms remains reasonable in practical cases: a daily run of the algorithms is in all cases possible.

Then, we present the insights we have obtained on the structure of the underlying graphs. First we note that the underlying graph of a F/OSS distribution can be generated in different ways, and that the method of generation changes the characteristics.

We also talk about the small world properties of the underlying graphs, and discuss the ramifications for the structure of the graph. It turns out that the graph of a distribution has a distinct structure: there are few packages with many dependencies, and many packages with just a few dependencies. The packages with many dependencies again fall into two distinct categories: high-level packages with many outgoing dependencies (but few or no incoming dependencies), and low-level packages with many incoming dependencies (but few or no outgoing dependencies). See also figure 8.3 on page 115.

1.3   Structure

The contents of the thesis are as follows: first, in chapter 2, we shall present an overview of the basics of Linux distribution management: its most-used package formats (Debian and RPM), and we shall recall the formalisation of F/OSS distributions devised in the EDOS project [MBDC+06, DCMB+06].

In chapter 3, we shall extend this formalisation with the concepts of *strong dependencies* and *strong conflicts*, as well as an application of *dominators* (a concept already known from flow control graphs) which allow for more extensive quality control over distributions. We shall also specify and prove some theorems that allow for more efficient computation of these concepts.

Then, in chapter 4, we present algorithms that use these theorems to efficiently compute strong dependencies, strong conflicts and dominators over a distribution. The actual implementation of these algorithms during the EDOS and MANCOOSI projects will be discussed in chapter 5.

The formalisation in Coq of the definitions and theorems from chapters 2 and 3 is the subject of chapter 6.

In chapter 7, we present the results of several experiments that have been executed using the tools from chapter 5. These results offer insights into the structure of the distributions; in chapter 8, we continue on this subject by discussing distributions when seen as graphs—this view offers other insights into the structure of distributions that can aid in managing them.

Finally, in chapter 9, I discuss the relevance of the subjects presented in this thesis, as well as related work and possible directions for future research.

1. Introduction

Definitions 2

'And why is it called the Carrock?' asked Bilbo as he went along at the wizard's side.
'He called it the Carrock, because carrock is his word for it. He calls things like that carrocks,
and this one is the Carrock because it is the only one near his home and he knows it well.'
— J.R.R. TOLKIEN, *The Hobbit*

In this preliminary chapter, we shall first specify in detail the package formats currently used for most Linux distributions: the Debian format and RPM. These two formats are very different in syntax; the difference in semantics, however, is much smaller, which is why it has been possible during the MANCOOSI project to devise a common format, called CUDF [TZ09], to which both formats can be translated.

After this, we shall discuss a package format from a different environment: the metadata format used for Eclipse plugins. Even though the environment in which this format is used is very different from RPM or the Debian format, we shall see that the basic metadata contents and semantics remain the same.

Having thus presented the existing package formats, we shall recall the definitions proposed in the EDOS project [DCMB+06]. These definitions are intended to be usable as a way of reasoning about any F/OSS distribution: they are sufficiently abstract to be used to represent packages from the Debian format, from RPM and even from Eclipse.

The main object of the EDOS formalisation is to reason about package installability. The effect of this is that a large part of the package metadata can be ignored, because it has no influence on installability. Examples of this are data like the name of the package maintainer, the package description or the package classification.

2.1 Existing package formats

2.1.1 Basic ideas

As discussed in the introduction, in F/OSS distributions, there exist interrelationships between packages. There are two main types: *dependencies* and *conflicts*. There are other types of relationships, but these are either equivalent to dependencies or conflicts, or can be safely ignored as far as installability is concerned.

For example, in Debian there is a *pre-dependency* relationship, which is like a normal dependency, except that it enforces an order of installation on the packages (a pre-dependency must be installed before configuration of the packages begins). Examples of relationships that can be ignored are the *recommendation* or *suggestion* relationships found in both Debian and RPM: these specify optional dependencies and thus do not influence installability.

A dependency relationship specifies that one package needs another to function; if package A depends on package B, the package manager takes care that when installing package A, package B also will be installed (but not the other way around).

Sometimes, it is possible to have *alternative* (or *disjunctive*) dependencies: in this case, a package can specify a list of other packages, of which at least one must be installed (it is allowed to have more than one package from the list installed as well; the disjunction is not exclusive).

A conflict specifies that one package cannot function when another package is installed. If package A conflicts with package B, the package manager takes care that package A is never installed at the same time as package B. Unlike the dependency relationship, the conflict relationship is symmetric (at least for Debian, RPM and Eclipse).

An example is shown in figure 2.1. The usual term for such a set of packages with dependency and conflict relations is a *repository*. Usually, there are multiple repositories available for one distribution: for example, the Debian distribution provides the `stable`, `testing` and `unstable` repositories. The difference between these three is that `stable` is, as the name suggests, very stable, but not up-to-date, whereas `unstable` is very up-to-date, but not always stable; `testing` is between the two.
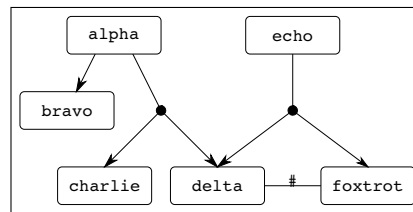

Figure 2.1: Example repository

In this case, the package `alpha` depends conjunctively on `bravo`, and disjunctively on `charlie` and `delta`. In order to install `alpha`, therefore, `bravo` must be installed, as well as either `charlie` or `delta` (or both). For the package `echo`, either `delta` or `foxtrot` needs to be installed. It is not possible to install both `delta` and `foxtrot`, because there is a conflict between them.

Many systems contain *virtual* packages. These are packages that do not physically exist, but can be *provided* by other packages. A dependency on a virtual package can be satisfied by any of the packages that provide it; a conflict with a virtual package means a conflict with all of the packages that provide it.

An example of this would be a web server package in an operating system; it would be a virtual package, provided by a number of specific web servers (Apache, lighttpd, …).

A simple example might make the idea more clear. In figure 2.2, the package `zulu` is a virtual package, which is provided by `bravo`, `charlie` or `delta`. Now, in order to install `alpha`, which depends on `zulu`, at least one of `bravo`, `charlie` or `delta` will be installed. On the other hand, in order to install `echo`, which *conflicts* with `zulu`, *none* of `bravo`, `charlie` or `delta` can be installed.

Most package systems use two different pieces of software for package management: the *installer* and the *meta-installer*.
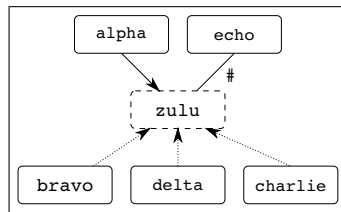
Figure 2.2: Example repository

The *installer* is usually responsible for installing and removing packages from the system, and keeping track of the installed packages and their files. It does not as a rule resolve dependencies or download packages; this is the task of the *meta-installer*, which also communicates with the user.

The general mode of operation of such a system is that the user requests an operation (install a package, upgrade every package in the system, . . .) from the meta-installer, which either decides that the request cannot be honoured (the user wants to install two conflicting packages, for example) or determines which packages have to be installed or removed to satisfy the request. The actual installation and/or removal of packages are subsequently executed by the installer.

2.1.2   The Debian format

The Debian package format (also called `.deb`) is defined in chapters 3–7 of the project's Policy Guide [DG98]. Its installer is called `dpkg`, and the standard meta-installer is `apt`. Another meta-installer called `aptitude` is also frequently used.

There is a distinction within Debian between *source packages* and *binary packages*. Binary packages are the ones installed on a user's machine; they are generated from source packages. A source package usually is the base for multiple binary packages; not only for the different architectures supported by Debian, but also for different options or parts of the software being packaged. As an example, the `ocaml` source package of the lenny distribution has 11 binary packages, such as `camlp4` (a preprocessor packaged with OCaml), `ocaml-base` (the OCaml base compiler and libraries), and `ocaml-base-nox` (the same, but without X11 libraries).

File contents

Debian binary packages follow the `.deb` format, defined in its manpage [DP06].
   A `.deb` file is an ar archive that contains at least three files:

1. a text file named `debian-binary` that contains the version number of the package; currently `2.0`.

2. a gzipped tar file named `control.tar.gz` that contains a set of text files with the package metadata. Of these, the file `control` is mandatory; it contains the core metadata.

3. a gzipped tar file named `data.tar.gz` that contains the files that belong to the package.

In order to easily allow access to the metadata of all packages in a distribution, there is also a file format that consists of the concatenation of `control` files from an entire distribution (as a text file). This type of file will from now on be referred to as a `Packages` file.

Package metadata

The package control file contains one or more paragraphs of fields, separated by blank lines. Each paragraph contains a list of fields: the field name, followed by a colon, followed by the field value. A field value may span several lines, in which case the second and following lines start with a space or tab. Other whitespace is ignored. Figure 2.3 is an example of the package control file for the `ocaml` package:

For a binary package, the `Package`, `Version`, `Architecture`, `Maintainer`, and `Description` fields are mandatory; `Section` and `Priority` are recommended. A list of the different fields and their meanings (for binary packages) follows:

Maintainer The name and e-mail address of the package maintainer.

Section This field is used for package classifications, as defined in the Policy Manual, Section 2.4.

Priority Package priority, as defined in the Policy Manual, Section 2.5.

Package The name of the package.

Architecture The architecture the package is intended for. If the value `all` is specified, the package is architecture-independent.

Essential If this field is set to `yes`, then the package is considered to be indispensable for a functioning system, and it should be installed at all times.

Depends This field specifies the dependency relationship mentioned in the previous section; its syntax will be explained in more detail below.

Pre-Depends This field specifies a special sort of dependency; it means that the package depended on must be installed *before* the package specifying the dependency.

Recommends This field specifies "a strong, but not absolute, dependency". The standard Debian package manager `apt` installs recommended packages by default.

Suggests This field specifies a weaker sort of dependency than a `Recommends` dependency; suggested packages are not installed by default by `apt`.

Enhances This field specifies the same sort of dependency as the `Suggests` field, but in the opposite direction.

Conflicts This field specifies a conflict between two packages, as explained in the previous section; the Debian package manager refuses to install two packages together if there is a conflict between them.

Breaks This field specifies a special, slightly weaker, kind of conflict: packages that break each other can be physically present on the system at the same time, but must not both be active ("configured").

Provides This field specifies that the package provides a virtual package, as explained in the previous section.

Replaces This field has two distinct uses:

- Normally, two packages cannot share the same file. However, if one package replaces the other, the Debian package manager will replace the file from the old package by the file from the new package.
- If two packages conflict with each other, and one of the packages is specified as replacing the other, instead of refusing to install them both, the Debian package manager will remove the replaced package and install the replacing package.

Version The package version.

Description A description of the contents of the package.

Installed-Size The estimated size of the package when installed, in kilobytes.

Homepage The URL for the home page of the software packaged.

Package interrelationships

The syntax of the package interrelationship fields is as follows: for the `Depends`, `Pre-Depends`, `Suggests`, and `Recommends` fields, they are a comma-separated list of alternatives; an alternative is a list of package names separated by a pipe symbol (|), optionally restricted to a version interval. Let us look at the dependency line for `ocaml`:

```
ocaml-base (= 3.10.2-3), ocaml-nox (= 3.10.2-3), libx11-dev
```

We see that there are three alternatives of exactly one package each (which means that all these three packages must be installed); for two packages, `ocaml-base` and `ocaml-nox`, it is specified that they must be installed with the exact version `3.10.2-3`.

Another, more complicated dependency line for the `abcde` package:

```
cd-discid, wget, cdparanoia | cdda2wav,
vorbis-tools (>= 1.0beta4-1) | lame | flac | bladeenc | speex
```

This package will install the packages `cd-discid`, `wget`, either `cdparanoia` or `cdda2wav`, and one (or more) of `vorbis-tools` (with a version higher than or equal to `1.0beta4-1`), `lame` (any version), `flac` (any version), `bladeenc` (any version) or `speex` (any version).

The syntax for the other interrelationship fields (`Conflicts`, `Breaks`) is similar, except that alternatives are not allowed here, the values are just a comma-separated list of package names, with eventual version restrictions.

Finally, the `Provides` field is just a comma-separated list of names, without any version specification.

Version comparison algorithm

A Debian package version number consists of three parts: the `epoch`, the version proper and, optionally, the `revision`. When comparing two versions, these three are compared in order: the epochs first, then the versions proper if there is no difference between the epochs, and finally the releases if there is no difference between the versions.

The epoch is an unsigned integer. If not specified, it is assumed to be 0.

The version proper (separated from the epoch by a colon) is an alphanumeric string that can additionally contain the characters ., +, -[1], :, and ~. If necessary (i.e. if the epochs are equal), these strings are compared by algorithm 1.

Algorithm 1 Compare the version strings v_1 and v_2, Debian style

---

> **while not** empty(v_1) **and not** empty(v_2) **do**
>   (n_1, v_1) ← non_digit_prefix(v_1)
>   (n_2, v_2) ← non_digit_prefix(v_2)
> r ← compare_lex(n_1, n_2)
>   **if** $r = 0$ **then**
>   (d_1, v_1) ← digit_prefix(v_1)
>   (d_2, v_2) ← digit_prefix(v_2)
>   **if** $d_1 < d_2$ **then** {Numerical comparison, empty string equivalent to 0}
>   **return** -1
>   **else if** $d_1 > d_2$ **then**
>   **return** 1
>   **end if**
>   **else**
>   **return** r
>   **end if**
> **end while**
> **return** 0

Here, `non_digit_prefix` and `digit_prefix` return a pair p, r where p is the prefix of the argument string that contains no digits (for `non_digit_prefix`) or only digits (for `digit_prefix`), and r the remainder of the argument string.

The function `compare_lex` compares two strings lexicographically, so that all letters sort before all letters, letters between them sort by ASCII value, and the tilde sorts before anything (even the empty string). For this and all other comparison algorithms discussed in this chapter, the convention is that the algorithm returns 0 if the strings are equal, -1 if the first argument is smaller, and 1 if the second argument is smaller.

The revision number (separated from the version by a hyphen) is an alphanumeric string that can additionally contain the characters +, . and ~. These

---

[1]The hyphen may only be present in the version proper if there is a revision number present.

strings are only compared if both the epochs and versions proper are equal; the algorithm is the same as that used for the version proper.

The rationale behind this version numbering scheme (which, as we shall see, is very similar to the one used for RPM) is that the version specified by the original author of the software should become the version of the package; this comparison algorithm works well with most versioning schemes used in practice.

However, it is possible that the original author decides to change the version numbering scheme (for example, to pass from a date-based scheme to a more classic $x.y.z$ scheme). In such a case, to make sure that the new version is deemed larger than the old version, the epoch can be raised.

The release part is used by the distribution to be able to make changes to the package, while keeping the same version of the original software.

Special semantics

In Debian, virtual packages do not have versions; hence, a dependency with a version restriction can *never* be satisfied by a virtual package.

Furthermore, a package can *never* conflict with itself. Thus, a special case occurs when a package both provides a virtual package and conflicts with it. Suppose a package has both `Provides: mail-transport-agent` and `Conflicts: mail-transport-agent` in its metadata. In this case, the package conflicts with any other package providing the same virtual package, in effect specifying that it should be the *only* package installed that provides `mail-transport-agent`.

If the package also has `Replaces: mail-transport-agent` in its metadata (in addition to the `Provides` and `Conflicts`) mentioned above, any package also providing `mail-transport-agent` will be removed by the Debian package manager.

In Debian, two versions of the same package implicitly conflict with each other, which means that it is not possible to have two packages that have the same name installed at the same time.

Installer and meta-installer

As said before, the tools used to install Debian packages are `dpkg`, the installer, and `apt`, the meta-installer.

In this section, we shall note some salient facts about `apt` and its most interesting part (for our purposes, anyway), the dependency solver. More extensive information on `apt` can be found in [DCMB+06].

Let us note first that the problem that `apt` tries to solve is indeed quite complicated, even beyond simple installability. In fact, `apt` has to deal with an existing system, one or more available distributions, and try to execute user requests while trying to maintain the system in a consistent state.

As we shall show later on, the installation problem itself is NP-complete. Since this means that dependency solving can take a very long time, `apt` does not try to be complete: if at some point in the calculation multiple options present itself (which is the case when a disjunctive dependency is encountered, none of whose packages are already installed), it just takes the first package from the list of alternatives and tries to install that. No backtracking is attempted.

In fact, this behaviour of `apt` has become something of a de facto standard, since it allows package managers to specify a preference in alternative dependencies: the package specified first in the alternative will be installed, unless another package from the alternative is already installed.

Furthermore, if multiple versions of the same package are available, only the highest version is installed; the idea being that the highest version of any package should always be the most advanced and bug-free one.

This can, of course, lead to false responses; imagine the following repository:

```
Package:  alpha
Version:  1.0
Depends:  bravo | charlie

Package:  bravo
Version:  1.0
Depends:  delta

Package:  charlie
Version:  1.0
```

The package `alpha` is perfectly installable when one uses `charlie` to satisfy its dependency. However, since `apt` only tries `bravo`—which is not installable, since its dependency `delta` is missing—it will return with an error.

The choice made by the developers of `apt` is to prefer a fast response over a correct one. Since the contents of the distribution are controlled by Debian, this does in general not result in too many errors, especially since most users use only one repository (`stable`, `testing` or `unstable`), which means that in most circumstances, only one version of every package is available, so that there are few disjunctive dependencies.

2.1.3 The RPM format

There is no authoritative specification of the syntax and semantics that we are aware of; most of the information in this section comes from experience, looking at the source of `rpm`, discussion with RPM developers, and [Bai97].

The installer for RPM is the `rpm` program[2], and the meta-installer used varies with the distributions; Mandriva uses `urpmi`.

File format

An RPM file is a binary file, divided into four parts:

1. The lead, unused in current implementations except to identify the file as an RPM file.

2. The signature, used to verify the integrity of the RPM file.

3. The header, which contains a list of `tags` that contain the metadata for the package.

---

[2]To distinguish between the package format and the installer, we shall use a normal font (RPM) for the package format, and teletype (`rpm`) for the installer.

4. The archive, which is a compressed `cpio` archive containing the package files.

Another file format is the `hdlist`, which is a concatenation of the headers of several packages (similar to the Debian `Packages` file).

Mandriva's `urpmi` meta-installer uses a textual format, called *synthesis* `hdlist`, which fulfils the same purpose as a 'normal' `hdlist`. However, it does not contain all tags from the RPM header and therefore it is far smaller (for Mandriva 2010.0, the `hdlist` is 149 Mb in size, whereas the synthesis `hdlist` is 3.8 Mb).

An example of the data in the synthesis `hdlist` for the `ocaml` package is in Figure 2.4.

Package metadata

The RPM metadata consists of a list of *tags*, with associated data. These are comparable to Debian's fields and values. Important tags are:

RPMTAG_NAME The package name.

RPMTAG_VERSION

RPMTAG_RELEASE

RPMTAG_EPOCH The package version, release and epoch.

RPMTAG_REQUIREFLAGS Flags for dependencies

RPMTAG_REQUIRENAME List of dependencies

RPMTAG_REQUIREVERSION Version requirements for dependencies

RPMTAG_CONFLICTFLAGS Flags for conflicts

RPMTAG_CONFLICTNAME List of conflicts

RPMTAG_CONFLICTVERSION Version requirements for conflicts

RPMTAG_DIRINDEXES

RPMTAG_BASENAMES

RPMTAG_DIRNAMES These three tags together contain the list of files for the package.

Version comparison algorithm

The version comparison algorithm used by `rpm` (which has been used in all the MANCOOSI tools that deal with RPM packages) works along the same lines as Debian's, but there are some important differences.

Like Debian, RPM version numbers consist of an epoch, a version proper and a release. These are compared in the same order as Debian's, and like for Debian, the first difference is decisive. However, there is one difference: in RPM, when comparing two versions of which one has a revision number, but the other does not, the revision numbers are ignored. This means that for example `1.27-1` is smaller than `1.27-2`, but both are equal to `1.27`.

The epoch is an integer, compared normally.

For comparison of the version proper and the release, the `rpmvercmp` function is used, with the behaviour specified in algorithm 2.

The `segment` function mentioned in algorithm 2 is used to split a version string into *segments*. A *segment* of a string is the longest prefix that either consists completely of numbers or consists completely of letters. It returns a pair (s, r): the segment and the remainder of the string.

The `compare_segment` function compares two segments: numerical segments are compared by value, while alphabetic segments are compared lexicographically. If one segment is numerical and the other is alphabetic, the numerical segment is considered to be smaller.

Algorithm 2 Compare the version strings v_1 and v_2, RPM style

---

  if $v_1 = v_2$ **then** {literal string comparison}
    return 1
  else
    while not empty(v_1) **and not** empty(v_2) **do**
      remove initial non-alphanumeric characters from v_1 and v_2
      $(s_1, v_1) \leftarrow$ segment(v_1)
      $(s_2, v_2) \leftarrow$ segment(v_2)
      if empty(s_1) **then**
        return -1 {s_2 cannot be empty (cf. the loop condition, so it must be greater)}
      else if empty(s_2) **then**
        return 1
      else
        $r \leftarrow$ compare_segments(s_1, s_2)
        if $r \neq 0$ **then**
          return r
        end if
      end if
    end while
    if empty(v_1) **then**
      if empty(v_2) **then**
        return 0
      else
        return -1
      end if
    else
      return 1
    end if
  end if

Special semantics

One very important difference in semantics between Debian and RPM is that there are no *direct* dependencies between packages in RPM. Any and all relationships between packages are realised using virtual packages. A package

always provides a virtual package with the same name and version as itself.

Another difference is that in RPM it is possible to attach a version specification to a virtual package; a package can thus specify that it provides only specific versions of a virtual package. Resolution is done by overlapping intervals: a package that provides virtual package A with version specification > 3 matches a package that requires virtual package A with version specification < 4, since they have the interval $< 3, 4 >$ in common.

In an RPM package, it is possible to specify a dependency on a file. This is handled differently by different pieces of software; `rpm` considers the dependency satisfied if the file exists on the file system; `urpmi` on the other hand treats file dependencies as normal dependencies (the relevant virtual packages are added during RPM generation).

Like Debian packages, RPM packages in principle cannot share the same file. However, there are flags in the RPM format that can modify this behaviour (notably in the case of config files: in this case, depending on options given by the user during installation, the file is either overwritten or backed up).

Installer and meta-installer

There are many different meta-installers for RPM, and they all have different behaviours. In this section, we shall use Mandriva's `urpmi` as an example.

Like with `apt`, the most interesting thing about `urpmi` is that it does not try to be complete in dependency solving, for the same reasons as `apt`.

The openSUSE meta-installer, however, uses a SAT solver implementation, `libzypp`, for dependency resolution [Sch08].

2.1.4   The Eclipse format

Eclipse is a very extensive integration platform for software development tools that, over the years, has accumulated a large amount of very diverse plugins [CR08].

As plugins became more complex and dependencies between plugins began to appear, the standard 'Update Manager' was no longer sufficient and the p2 project [LBR10] was started to develop a more satisfactory plugin management system. We shall discuss the properties of this system, which resembles Debian and RPM, though there are many differences as well, in this section; much of the data in this section comes from [Boz10].

Let us first note that the p2 system operates in a different environment than `apt` and `rpm`; instead of managing a complete OS and its software, p2 operates within the context of the Eclipse system and its plugins. Nonetheless, the familiar concepts of packages, dependencies and conflicts are all present in p2 as well.

Package metadata

To start with, the basic unit that is normally known as a package is called an IU (*installable unit*) in the p2 universe. It is uniquely identified by a string (the identifier), and a version (consisting of three integers a string, for example something like `2.1.0.beta`).

Dependencies between IUs are created by *requirements* and *capabilities*: as with RPM, an IU *A* depends on another IU *B* if *A* has a requirement that matches a capability of *B*.

Furthermore, every IU has an *enablement filter*, with which it can specify conditions it needs to be installed, for example a particular OS or architecture.

IUs also have the possibility to set a *singleton flag* to specify that the system should not contain another IU with the same identifier (this is somewhat like the combination of `Provides` and `Conflicts` fields in Debian).

And finally, an IU has an *update specification*, in which it identifies the IUs that are considered predecessors to this IU.

Capabilities and requirements, as mentioned before, are used to create dependencies between packages. A capability consists of a namespace, a name (both strings), and a version; a requirement consists of a namespace, a name and a version range. A requirement matches a capability if the namespace and name match, and the version of the capability is included in the version range of the requirement.

In addition to this, a requirement has a filter which can result in its being disabled under certain conditions (see the enablement filter for IUs mentioned above).

Finally, requirements have two additional properties: they can be *optional* and *greedy*. An optional requirement, as the name indicates, is a requirement that does not have to be satisfied for the IU to be installed; the 'greed' property indicates whether an IU satisfying the requirement must be actively sought (greedy) or whether it must just be verified that the requirement has been met (non-greedy).

In fact, a requirement that is optional and non-greedy need not be installed at all; it is therefore akin to the `Suggests` field in the Debian format. If an IU satisfying the requirement is found, it will be added to the dependencies of the IU to be installed, but if no such IU is found, the requirement will simply be considered satisfied and no further action will be taken.

Version comparison algorithm

The version comparison algorithm is like those used for Debian and RPM, but much simpler. As mentioned above, an Eclipse version number consists of three integers (major version, minor version and micro-version) and a string.

To compare two version numbers, first the two major versions are compared numerically. If there is a difference, the version number with the highest major version is the highest version number.

Otherwise, the minor versions are compared in the same way, followed by the micro-versions.

Finally, the two strings are compared with the standard Java string comparison function; the result of this comparison becomes the result of the entire version number comparison.

2.2 Definitions

In this section, we shall recall the formalisation of 'package theory' devised in the EDOS project [MBDC⁺06, DCMB⁺06]. The intent of this formalisation is

to reflect the features of both standards presented above, especially those used to determine package installability. We shall also present several lemmas that follow easily from these definitions, and that will come in useful when defining more complicated properties later on.

The atomic entity in this thesis is the *package*; we shall abstract away from names and version numbers, since there are no theorems in this thesis that make use of these properties.

Definition 2.1 (Repository)
A repository (R, D, C) is a triple consisting of a set of packages P, a conflict relation C ($C \subseteq R \times R$), and a dependency function $D : R \longrightarrow \wp(\wp(R))$.

Axiom 2.2
For any package $p \in R$, there does not exist a $d \in D(p)$ such that $p \in d$.

In other words, a package never depends directly upon itself. This is explicitly stated in the Debian specification, though not in the RPM specification. But even so, such a dependency would be trivially fulfilled (in order to install a package, the package itself is always installed); thus, it is safe to forbid such dependencies.

Axiom 2.3
The conflict relation is symmetrical and irreflexive.

Thus, a package can never conflict with itself. The Debian method of specifying a conflict with a virtual package will be treated later on.

As for the dependency function, it associates a set of *alternatives* with a package. These alternatives (which are themselves sets of packages) represent disjunctive dependencies.

For example, the representation of the distribution from figure 2.1 would be:

$$
\begin{aligned}
R &= \{\texttt{alpha}, \texttt{bravo}, \texttt{charlie}, \texttt{delta}, \texttt{echo}, \texttt{foxtrot}\} \\
D(\texttt{alpha}) &= \{\{\texttt{bravo}\}, \{\texttt{charlie}, \texttt{delta}\}\} \\
D(\texttt{bravo}) &= \emptyset \\
D(\texttt{charlie}) &= \emptyset \\
D(\texttt{delta}) &= \emptyset \\
D(\texttt{echo}) &= \{\{\texttt{delta}, \texttt{foxtrot}\}\} \\
D(\texttt{foxtrot}) &= \emptyset \\
C &= \{(\texttt{delta}, \texttt{foxtrot}), (\texttt{foxtrot}, \texttt{delta})\}
\end{aligned}
$$

In this formalisation, we do not consider virtual packages. One can in fact consider a dependency on a virtual package as a giant disjunction, and a conflict with a virtual package as a conflict with any package providing that package. For example, consider the following Debian repository:

```
Package: alpha
Version: 1.0
Provides: zulu

Package: bravo
Version: 1.0
Provides: zulu

Package: charlie
Version: 1.0
Depends: zulu

Package: delta
Version: 1.0
Conflicts: zulu
```

This can be represented as:

$$
\begin{aligned}
R &= \{\texttt{alpha}, \texttt{bravo}, \texttt{charlie}, \texttt{delta}\} \\
D(\texttt{alpha}) &= \emptyset \\
D(\texttt{bravo}) &= \emptyset \\
D(\texttt{charlie}) &= \{\{\texttt{alpha}, \texttt{bravo}\}\} \\
D(\texttt{delta}) &= \emptyset \\
C &= \{(\texttt{alpha}, \texttt{delta}), (\texttt{delta}, \texttt{alpha}), (\texttt{bravo}, \texttt{delta}), (\texttt{delta}, \texttt{bravo})\}
\end{aligned}
$$

In essence, the virtual package is replaced by the packages that provide it. It is possible that a package simultaneously provides and conflicts with the same virtual package; in Debian semantics, this means that the package wants to be the only provider of the virtual package that is installed. This mechanism can be simulated by replacing the provide/conflict specification with a conflict with every other provider.

2.3   Installability

For a package p to be installable, with respect to a given repository (R, D, C), it must be possible to find a set I of packages in the repository (the *installation*; $I \subseteq R$) that contains the package and fulfil two conditions: all the dependencies in I must be satisfied and no two packages in I are in conflict.

These conditions are called *abundance* and *peace* respectively. The combination of both is referred to as *health*.

Definition 2.4 (Abundance)
A set of packages I is abundant (with respect to a repository (R, D, C)) if and only if:

$$\forall_{p \in I}[\forall_{d \in D(p)}[I \cap d \neq \emptyset]]$$

Note that abundance (and hence installability) is always defined with respect to a given repository: a package may be installable in one repository, but not in an other one.

Corollary 2.5
If two sets of packages I_1 and I_2 are abundant with respect to a repository (R, D, C) , their union $I_1 \cup I_2$ is abundant with respect to the repository (R, D, C).

Definition 2.6 (Peace)
A set of packages I is peaceful (with respect to a repository (R, D, C)) if and only if:

$$\forall_{(c_1, c_2) \in C}[\neg(c_1 \in I \wedge c_2 \in I)]$$

Like abundance, peace is always defined with respect to a specific repository.

Definition 2.7 (Health)
A set of packages is healthy with respect to a repository (R, D, C) if it is abundant and peaceful with respect to (R, D, C).

In the following, to make the notation easier to read, we shall omit the repository (R, D, C) when using the above properties, when it is clear from the context which repository is intended.

With these definitions, it becomes possible to formalise the *installability* of a package:

Definition 2.8 (Installability)
A package p is installable *in a repository (R, D, C) if and only if there exists a healthy set $I \subseteq R$ such that $p \in I$.*

The set I is also called an *installation set* of I; note that there are in general several possible installation sets for a given package. For example, when we look at the repository from figure 2.1, possible install sets for `alpha` are $\{$`alpha`, `bravo`, `charlie`$\}$ and $\{$`alpha`, `bravo`, `delta`$\}$. These are just the minimal sets, in fact: $\{$`alpha`, `bravo`, `charlie`, `delta`, `echo`$\}$ also is a perfectly valid install set (it is not necessary to install both `charlie` and `delta`, nor `echo`, but

neither is it forbidden to install extra packages) However, since an install set has to be peaceful, it is not possible to include both `delta` and `foxtrot` in an install set, because of the conflict between them.

This definition is easy to extend to multiple packages; if several packages are installable at the same time, they are co-installable:

Definition 2.9 (Co-installability)
A set of packages P is co-installable in a repository (R, D, C) if and only if there exists a healthy set $I \subseteq R$ such that $P \subseteq I$.

2.4 Dependencies

A distribution can also be seen as a graph, where the packages are the vertices and dependencies are the edges.

One can use special nodes to represent disjunctions, to help visualise the structure of the dependencies, like in figure 2.1, but it is also useful to consider the graph obtained when forgetting all differences between conjunctive and disjunctive dependencies, and the graph obtained when using only conjunctive dependencies. More about distributions when considered as graphs can be found in chapter 8.

Definition 2.10 (Direct dependency)
A package p depends directly *on another package q ($p \rightarrow q$) if and only if there is a d in $D(p)$ such that $q \in d$.*

This can be made into a graph (V, E) where $V = R$ and $E = \{(p, q) \mid p \rightarrow q\}$. If necessary, a function $f : E \longrightarrow \{\texttt{Conjunctive}, \texttt{Disjunctive}\}$ can be added to mark edges as representing conjunctive or disjunctive dependencies.

The interest of such a graph is that it allows one to visualise which packages can have a potential effect on one another, even though an edge between two packages does not always indicate that installing one also means installing the other.

Definition 2.11 (Dependency)
A package p depends on another package q ($p \twoheadrightarrow q$) if and only if there is a list of packages $x_0, x_1, \ldots, x_n$ such that $p \rightarrow x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n \rightarrow q$.

Corollary 2.12
The dependency relation is transitive.

In order to keep the difference between conjunctive and disjunctive dependencies, if desired, one can add the following definitions:

Definition 2.13 (Conjunctive direct dependency)
A package p has a conjunctive *direct dependency on a package q ($p \xrightarrow{c} q$) if and only if there is a d in $D(p)$ such that $d = \{q\}$.*

Definition 2.14 (Conjunctive dependency)
A package p has a conjunctive *dependency on a package q ($p \xrightarrow{c}\!\!\!\!\twoheadrightarrow q$) if and only if there is a list of packages $x_0, x_1, \ldots, x_n$ such that $p \xrightarrow{c} x_0 \xrightarrow{c} x_1 \xrightarrow{c} \ldots \xrightarrow{c} x_n \xrightarrow{c} q$.*

It now becomes possible to define the *dependency cone* of a package.

Definition 2.15 (Dependency cone)
The dependency cone $\Delta_R(p)$ of a package p with respect to a repository (R, D, C) is the set of packages $\{q \in R \mid p \twoheadrightarrow q\}$.

Similarly, the dependency cone $\Delta_R(P)$ of a set of packages P is the union $\bigcup_{p \in P} \Delta_R(p)$ (or, equivalently, $\{q \in R \mid \exists_{p \in P}[p \twoheadrightarrow q]\}$).

When it is clear from the context which repository is meant, we shall simply write $\Delta(p)$ or $\Delta(P)$.

Constructing the dependency cone is equivalent to taking the transitive closure of the direct dependency relation.

Similar to the dependency cone, there is also the *reverse dependency cone* of a package:

Definition 2.16 (Reverse dependency cone)
The reverse *dependency cone $\overline{\Delta}_R(p)$ of a package p with respect to a repository (R, D, C) is the set of packages $\{q \in R \mid q \twoheadrightarrow p\}$.*

The importance of the dependency cone comes from the following property:

Proposition 2.17
A package p is installable with respect to a repository (R, D, C) if and only if it is installable with respect to the repository $(\Delta_R(p), D, C)$.

Proof In two directions:

- Suppose that p is installable w.r.t (R, D, C). Then, there is a healthy set $I \subseteq R$ with $p \in I$. Now $I \cap \Delta_R(p)$ also is healthy; it is abundant because $D(p) = D(p)|\Delta_R(p)$, and it is peaceful, because $I \cap \Delta_R(p) \subseteq I$, and there are no conflicting packages in I.

- Suppose that p is installable w.r.t. $(\Delta_R(p), D, C)$. Since $R \supseteq \Delta_R(p)$, one can re-use the same installation set for (R, D, C).

This means that, when determining the installability of a package (or set of packages) in a repository, one only has to consider packages that are in its dependency cone. This is not surprising, as the installability of a package cannot depend on packages that it has no dependency relation with, but it is of great importance when building efficient algorithms: many tools which are being used today to check installability are based on solvers that work on a propositional logic representation of a repository. The property proven in proposition 2.17 shows that the dependency solver only needs the packages in the dependency cone of the package it is investigating, which saves time and space.

Another important property is stated in the following proposition: if a package p has a conjunctive dependency on another, any installation set of p will contain this package.

Proposition 2.18
For two packages p, q such that $p \overset{c}{\twoheadrightarrow} q$, any installation set of p includes q.

Proof Since $p \overset{c}{\twoheadrightarrow} q$, there is a list $x_0, x_1, \ldots x_n$ such that $p \overset{c}{\to} x_0 \overset{c}{\to} x_1 \overset{c}{\to} \ldots \overset{c}{\to} x_n \to_c q$.

Follows a proof by induction on k that given an installation set I of p, for all x_k, $x_k \in I$:

- $x_0 \in I$: $p \overset{c}{\to} x_0$, so that $\{x_0\} \in D(p)$. Since I is abundant, $\{x_0\} \cap I \neq \emptyset$, hence $x_0 \in I$.

- $x_m \in I \longrightarrow x_{m+1} \in I$: $x_m \overset{c}{\to} x_{m+1}$, so that $\{x_{m+1}\} \in D(x_m)$. Since I is abundant, $\{x_{m+1}\} \cap I \neq \emptyset$, hence $x_{m+1} \in I$.

Thus, $x_n \in I$. Given that $x_n \overset{c}{\to} q$, $\{q\} \in D(x_n)$ and thus $q \in I$.

Again, this property seems trivial, but we shall see in later chapters that it can be used to optimise several algorithms.

```
Package: ocaml
Priority: optional
Section: devel
Installed-Size: 8368
Maintainer: Debian OCaml Maintainers <debian-ocaml-maint@lists.debian.org>
Architecture: i386
Version: 3.10.2-3
Replaces: ocaml-nox (« 3.10.0-12)
Provides: ocaml-3.10.2
Depends: ocaml-base (= 3.10.2-3), ocaml-nox (= 3.10.2-3), libx11-dev
Suggests: tcl8.4-dev, tk8.4-dev
Filename: pool/main/o/ocaml/ocaml_3.10.2-3_i386.deb
Size: 2066194
MD5sum: cd876d71c86a2ed80f052f2994745dd7
SHA1: 0a65ca56fa0fc56c24ad44aa0b73b06e0d000bd1
SHA256: 7dcc3186984741313c663b638c68cf9f33028e71154ace3e4f35c4c46f4148bb
Description: ML language implementation with a class-based object system
Objective Caml (OCaml) 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.
 .
OCaml comprises two compilers. One generates bytecode
which is then interpreted by a C program. This compiler runs quickly,
generates compact code with moderate memory requirements, and is
portable to essentially any 32 or 64 bit Unix platform. Performance of
generated programs is quite good for a bytecoded implementation:
almost twice as fast as Caml Light 0.7. This compiler can be used
either as a standalone, batch-oriented compiler that produces
standalone programs, or as an interactive, toplevel-based system.
 .
The other compiler generates high-performance native code for a number
of processors. Compilation takes longer and generates bigger code, but
the generated programs deliver excellent performance, while retaining
the moderate memory requirements of the bytecode compiler. It is not
available on all arches though.
 .
This package contains everything needed to develop OCaml applications,
including the graphics libraries.
Homepage: http://caml.inria.fr/
Tag: devel::{compiler,interpreter,lang:ocaml}, implemented-in::ocaml,
role::meta
```

Figure 2.3: Metadata for the Debian `ocaml` package

```
provides@ocaml-emacs@dllbigarray.so@dllgraphics.so@dllmldbm.so@dllnums.so@
dllstr.so@dllthreads.so@dllunix.so@dllvmthreads.so@libcamlrun_shared.so@
ocaml[== 3.11.0-2mdv2009.1]@ocaml(x86-32)[== 3.11.0-2mdv2009.1]
@obsoletes@ocaml-emacs
@requires@bash@libX11.so.6@libc.so.6@libc.so.6(GLIBC_2.0)@
libc.so.6(GLIBC_2.1)@libc.so.6(GLIBC_2.1.2)@libc.so.6(GLIBC_2.1.3)@
libc.so.6(GLIBC_2.2)@libc.so.6(GLIBC_2.3)@libc.so.6(GLIBC_2.3.2)@
libdb-4.7.so@libdl.so.2@libdl.so.2(GLIBC_2.0)@libdl.so.2(GLIBC_2.1)@
libm.so.6@libm.so.6(GLIBC_2.0)@libncurses.so.5@libpthread.so.0@
libpthread.so.0(GLIBC_2.0)@libpthread.so.0(GLIBC_2.2)@rtld(GNU_HASH)
@summary@The Objective Caml compiler and programming environment
@filesize@5868328
@info@ocaml-3.11.0-2mdv2009.1.i586@0@27475067@Development/Other
```

Figure 2.4: Synthesis data for ocaml in Mandriva 2010.0

Strong dependencies and conflicts 3

In F/OSS distributions, as well as many other component-based systems [Apa09, CR08], the language used to express inter-package relationships is expressive enough to cover propositional logic. As a consequence, considering only 'normal' dependencies, as expressed in the existing metadata, the existence of a dependency path between two packages does not guarantee that when installing the first package, the second will always be installed. For example, if p is to be installed and there exists a dependency path from p to q, it is not true that q is always needed for p, and in some cases q may even be incompatible with p.

In other terms, the *syntactic* connectivity notion—this being the existence of a dependency path as specified in the package metadata—does not tell us much about the real structure of dependencies and conflicts: it is necessary to go further and analyse the *semantic* connectivity—the essence of the dependency relationship: installing one package always implies installing another package as well—among software components induced by the explicit dependencies in the graph.

In this chapter, we shall explain these notions of *semantic* connectivity and propose their basic properties, as well as theorems that can be used to efficiently compute them. The consequences of this notion when considering the distribution graph will be treated in more detail in chapter 8.

3.1 Strong dependencies

When considering the dependencies of a package, it is interesting to restrict ourselves to those dependencies that are *always* installed when the package itself is installed; this gives us an under-representation of the actual packages that are going to be installed (there might still be other packages that are part of a disjunctive dependency, for example), but it does give us the most important dependencies: those that are absolutely essential.

In the MANCOOSI project, we have called this concept a *strong dependency*: in short, a package p depends strongly q if and only if it is impossible to install p without also installing q. The formal definition is as follows:

Definition 3.1 (Strong dependency)
Given a repository $\varrho = (R, D, C)$, a package p strongly depends on a package q (denoted $p \Rightarrow_\varrho q$) if and only if p is installable in ϱ, and all installation sets of p in ϱ contain q (If it is clear from the context which repository is meant, we shall simply note $p \Rightarrow q$ to indicate a strong dependency).

> *The set of strong dependencies of a package p, $\{q \mid p \Rightarrow q\}$, is denoted as $Scons(p)$ (the strong consequences of p).*

Note that for a package to have strong dependencies, it has to be installable; without this condition, every non-installable package would have trivial strong dependencies on every other package.

In figure 3.1, we see that conjunctive dependencies (such as `alpha` → `bravo` and `alpha` → `charlie`) translate to identical strong dependencies (with the proviso that the package must be installable), but that disjunctive dependencies (such as `delta` → `echo` or `delta` → `foxtrot`) do not.

(strong dependencies bold, normal dependencies dotted)
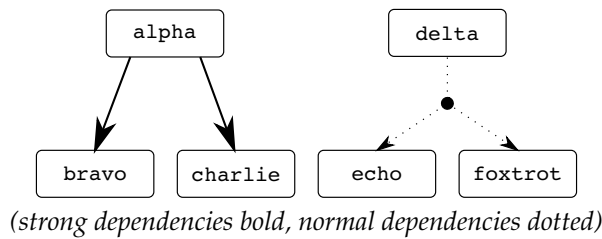
Figure 3.1: Simple example

The more complicated example in figure 3.2, shows that disjunctive dependencies *can* translate to strong dependencies (in this case, because of the conflict between `alpha` and `charlie`, `bravo` becomes a strong dependency, as it is the only way to satisfy the disjunctive dependency and will therefore always be installed if `alpha` is installed).
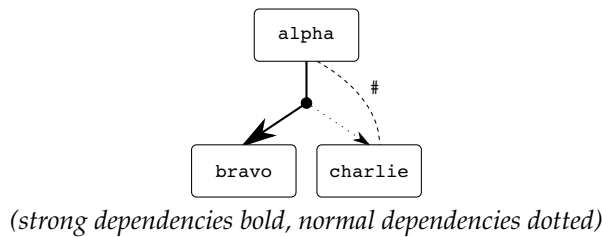

(strong dependencies bold, normal dependencies dotted)

Figure 3.2: More complicated example

This gives us our first corollary: conjunctive dependencies are strong dependencies.

> **Corollary 3.2**
> *If $p \xrightarrow{c} q$ and p, q are packages in ϱ, and p installable w.r.t. ϱ, then $p \Rightarrow_\varrho q$ (a conjunctive dependency implies a strong dependency).*

Another corollary:

> **Corollary 3.3**
> *The strong dependency relation is reflexive and transitive.*

In chapter 4, we shall discuss an efficient way to compute the strong dependencies within a repository.

The notions of strong and direct dependencies can be used to create graphs:

> **Definition 3.4 (Dependency graphs)**
> *The strong dependency graph $SG(\varrho)$ of a repository $\varrho = (R, D, C)$ is the directed graph with the elements of R as its vertices, and as edges all pairs p, q such that $p \Rightarrow_\varrho q$. (Note that since the strong dependency relation is transitive, $SG(\varrho)$ is closed under transitivity).*
>
> *Similarly, the direct dependency graph $DG(\varrho)$ of a repository $\varrho = (R, D, C)$ is the directed graph with the elements of R as its vertices, and as edges all pairs p, q such that $p \to q$.*

These graphs and their properties will be discussed further in chapter 8.

> **Definition 3.5 (Impact set)**
> *Given a repository $\varrho = (R, D, C)$, the impact set $Is(p, \varrho)$ of a package p is the set $\{q \in R \mid q \Rightarrow_\varrho p\}$.*

It now becomes easy to define the "sensitivity" of a package—a measure of how many other packages can be affected by a change in it.

> **Definition 3.6 (Sensitivity)**
> *The sensitivity of a package p in ϱ is defined as: $|Is(p, \varrho)| - 1$; in other words, the cardinality of its impact set minus 1. Because the impact set of a package always contains itself, 1 is subtracted; in this way, a package on which no other package strongly depends has a sensitivity of 0.*

Note that any installation set of a package p must necessarily include all strong dependencies of p. In other words:

> **Corollary 3.7**
> *For any installation set I of p, it holds that $Scons(p)\varrho \subseteq I$.*

We shall use this observation in chapter 4 to specify an efficient algorithm for the computation of the strong dependencies present in a distribution.

3.2 Dominators

When analysing a large component base, like Debian's, which contains about 22 000 components, it is important to be able to identify some measure that

can be used to easily pinpoint 'interesting' packages. Sensitivity can be (and actually is, in our tools) used to order packages, bringing the most sensitive to the forefront. But sensitivity alone is not enough: one does not want to spend time going through hundreds of packages with similar sensitivity to find the one which is really important, so some of the structure of the strong dependency graph should be conserved.

A first step is to group together only those packages that are related by strong dependencies, but analysis of the Debian distribution has shown that it is necessary to go further and distinguish the cases of related components in the strong dependency graph from the cases of unrelated ones: in the picture in figure 3.3 [1], configuration 3.3c shows quebec that clearly dominates romeo, as the impact set of romeo is actually the impact set of quebec, plus romeo itself. In the same vein, in configuration 3.3d, romeo and quebec are equivalent (their impact sets are equal, and they strongly depend on each other). Conversely, in configuration 3.3a, romeo and quebec are not immediately related to each other (other than that their impact sets overlap, but this only means that the same packages depend on them, and does not indicate any relation between the two packages), and in configuration 3.3b, quebec strongly depends on romeo, but a part of the impact of romeo has nothing to do with quebec.

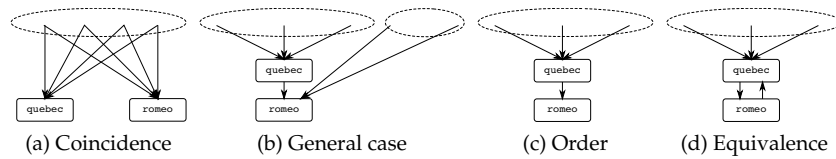| (a) Coincidence | (b) General case | (c) Order | (d) Equivalence |

Figure 3.3: Significant configurations in the strong dependency graph

Yet, the packages romeo and quebec all have essentially the same sensitivity values. To distinguish between these different configurations in strong dependency graphs, we shall introduce one last notion: dominance.

This notion is known from the domain of flow control graphs; there, a node p dominates another node q if and only if every path from the start node to q passes through p. Our definition looks different, but we shall show later on that it is equivalent to the notion of dominance from flow control graphs.

Another way of imagining dominance, which is more pertinent to distributions, is that p dominates q if and only if the impact set of p *explains* the impact set of q: the impact set of q consists of the impact set of p, plus some extra packages explained by the fact that q is a strong dependency of q. The formal definition follows:

Definition 3.8 (Dominance)
Given two packages p and q in a repository ϱ, p strongly dominates q ($p \succcurlyeq_{Is} q$) if and only if:

1. $Is(p, \varrho) \supseteq (Is(q, \varrho) \setminus Scons(p))$;

2. $p \Rightarrow q$

---

[1] Edges implied by transitivity are omitted for the sake of clarity

Using the transitivity of strong dependencies, it is possible to prove that the strong domination relation is a partial pre-order:

Lemma 3.9

The dominance relation is a partial pre-order.

Proof

- **Reflexivity**: trivial to check.

- **Transitivity**: suppose that there are p, q, r such that $p \succcurlyeq_{Is} q$ and $q \succcurlyeq_{Is} r$. Then, $Is(p, \varrho) \supseteq (Is(q, \varrho) \setminus Scons(p))$ and $Is(q, \varrho) \supseteq (Is(r, \varrho) \setminus Scons(q))$. By transitivity of strong dependencies, since $p \Rightarrow q \Rightarrow r$, it is also the case that $Scons(p) \supseteq Scons(q) \supseteq Scons(r)$. Then, $(Is(r, \varrho) \setminus Scons(q)) \setminus Scons(p) = Is(r, \varrho) \setminus Scons(p)$, and because $Is(p, \supseteq) Is(r, \varrho) \setminus Scons(q)$, it is the case that $p \succcurlyeq_{Is} r$.

  By transitivity of the strong dependency relation, it is also the case that $p \Rightarrow r$.

This pre-order is now able to distinguish among the cases of figure 3.3. In configuration 3.3c, it is the case that quebec $\succcurlyeq_{Is}$ romeo, but not the converse; in configuration 3.3d both quebec $\succcurlyeq_{Is}$ romeo and romeo $\succcurlyeq_{Is}$ quebec (in other words, romeo and quebec are equivalent with respect to dominance); in configurations 3.3a and 3.3b, no dominance relationship can be established between romeo and quebec.

It is possible, and actually quite useful, to generalise the dominance relation to also cover the case from configuration 3.3b, where a part of the impact set of the package romeo is not covered by the impact set of quebec.

If the "uncovered" part is small in respect to the "covered" part, after all, there is still a strong correlation between the impact sets of both packages. This concept of relative dominance is defined as follows:

Definition 3.10 (Relative dominance)

Given two packages p and q in a repository ϱ, p strongly dominates q up to z ($p \succcurlyeq_{Is}^{z} q$) if and only if:

- $\frac{|(Is(q, \varrho) \setminus Scons(p)) \setminus Is(p, \varrho)|}{|Is(p, \varrho)|} * 100 \leq z$;

- *p strongly depends on q.*

It is easy to see that $p \succcurlyeq_{Is} q$ is equivalent to $p \succcurlyeq_{Is}^{0} q$, and one can compute in a single pass on the repository the values z for each pair of packages such that $p \Rightarrow q$, leaving for later the choice of a threshold value for z.

3.3 Dominators in strong dependency graphs and control flow graphs

Dominance in a strong dependency graph can be formally put in correspondence with the traditional notion of dominators in control flow graphs [LT79].

This does require some manipulation of the graphs, though, because unlike regular control flow graphs, strong dependency graphs are transitives and do not have a start node.

The key idea is to build a control flow graph out of a strong dependency graph, by first performing transitive reduction, and then adding a start node that connects to every node that does not have any predecessors.

Before starting with the main proof, we introduce some auxiliary lemmas. To start with, since the strong dependency graph may contain cycles, the transitive reduction is not unique[LT79]. However, since the graph is closed under transitivity, all cycles are actually cliques, and all the vertices of such a cycle are equivalent to each other in the dominance relation:

Lemma 3.11 (Equivalence)
If $p \Rightarrow q$ and $q \Rightarrow p$, then $p \succcurlyeq_{Is} q$ and $q \succcurlyeq_{Is} p$

Proof If $p \Rightarrow q$ and $q \Rightarrow p$, then $Scons(p) = Scons(q)$, and $Is(p, \varrho) = Is(q, \varrho)$. Hence, $Is(p, \varrho) \setminus Scons(q) = Is(q, \varrho) \setminus Scons(p)$, and therefore $p \succcurlyeq_{Is} q$ and $q \succcurlyeq_{Is} p$.

Lemma 3.12
If there is a cycle $v_1 \to v_2 \to \ldots \to v_n$ in $SG(\varrho)$, then $\{v_1, v_2, \ldots, v_n\}$ is a clique in $SG(\varrho)$.

Proof To be proven is that for any v_i and v_j ($1 \le i, j \le n$), there is an edge between v_i and v_j. Since v_i and v_j are part of a cycle, there is a path from v_i to v_j, and because $SG(\varrho)$ is closed under transitivity, there must also be an edge from v_i to v_j.

Since all vertices in such cliques are equivalent, one can replace the entire clique by one vertex that represents its equivalence class in the strong dominance relation; the resulting graph does not contain any cycles. This is slightly more simple than the traditional approach to the transitive reduction of cyclic graphs, where one has to choose between the possible ways of expanding back this representative node into a cycle.

Definition 3.13 (Collapse)
Given a graph G, its collapse $G \downarrow$ is defined as the graph obtained by the following procedure:

- *For all maximal cliques $C = \{v_1, v_2, \ldots, v_n\}$, do*

    - *Add a vertex v_C to G.*
    - *Replace all edges $v \to v_i$ with $v \to v_C$, and $v_i \to v$ with $v_C \to v$.*
    - *Remove all edges $v_C \to v_C$.*

> – *Remove all vertices in C from G.*
>
> • *Perform transitive reduction on the resulting graph.*
>
> *The function $\varphi_G : V(G) \longrightarrow V(G \downarrow)$ is defined to map a node in G to its replacement in $G \downarrow$ (we shall just write φ if it is clear from the context with G is intended). Note that this function is a surjection.*

Lemma 3.14
If G is a transitive graph, then $G \downarrow$ is acyclic.

Proof Since G is transitive, by lemma 3.12, any cycle in G is a clique.

Now, it becomes possible to prove that *all* cliques are collapsed in $G \downarrow$.

By definition 3.13 all maximal cliques are replaced by a single node in $G \downarrow$.

Every clique is either a maximal clique itself, or a complete subclique of a maximal clique, or shares at least one vertex with another maximal clique.

It is obvious that by replacing all maximal cliques, all cliques that correspond to the first two cases are removed.

To conclude the proof, it remains to show that in a transitive graph cliques that correspond to the third case do not exist.

Suppose there is a maximal clique M, and a clique C that is not maximal, is not a complete subclique of M, but shares at least one vertex v with M. Since v is connected to all vertices in both C and M, by transitivity of G, every vertex in C must be connected to every vertex in M and vice versa. Hence, $M \cup C$ is a clique as well, which contradicts our assumption that M is a maximal clique.

All this allows to build the flow graph of strong dependencies:

Definition 3.15 (Flow graph)
Given a graph of strong dependencies $SG(\varrho)$, the corresponding flow graph of strong dependencies $FG(\varrho)$ is obtained from $SG(\varrho) \downarrow$ by adding an extra start *vertex which is connected to every vertex in $SG(\varrho) \downarrow$ that does not have any predecessor.*

Lemma 3.16
For every vertex $v \in FG(\varrho)$, there exists a path start $\twoheadrightarrow v$.

Proof Any vertex $x \in SG(\varrho) \downarrow$ (with $\varphi(x) = v$) must either have no predecessors (and thus, by construction, there must be an edge between start and v in $FG(\varrho)$), or have at least one predecessor. Since, by lemma 3.14, $SG(\varrho) \downarrow$ is acyclic, and ϱ is finite, this predecessor cannot be part of an infinite path; at

some point in the transitive closure of the predecessors of x, a node w must be encountered that has no predecessors. There is a path $\text{start} \to \varphi(w)$, and a path $\varphi(w) \twoheadrightarrow v$, and thus there is a path $\text{start} \twoheadrightarrow v$.

Lemma 3.17

A vertex $\varphi(w)$ is reachable from a vertex $\varphi(v)$ in $FG(\varrho)$ if and only if $v \Rightarrow_\varrho w$.

Proof

($\Leftarrow$) Let us assume that $v \Rightarrow_\varrho w$; then it must be proven that there is a path from $\varphi(v)$ to $\varphi(w)$ in $FG(\varrho)$.

Given that $v \Rightarrow_\varrho w$, there is an edge $v \to w$ in the graph $SG(\varrho)$. If neither v nor w are part of a clique in $SG(\varrho)$, there is a path $v \twoheadrightarrow w$ in the transitive reduction of $SG(\varrho)$ and therefore there is a path from $\varphi(v)$ to $\varphi(w)$ in $SG(\varrho) \downarrow$. This is also true if v and w are part of different cliques; by construction, the path from v to w is maintained between $\varphi(v)$ and $\varphi(w)$ $FG(\varrho)$.

If v and w are part of the same clique, both v and w are replaced in $SG(\varrho) \downarrow$ by the same node (and thus in $FG(\varrho)$, $\varphi(v) = \varphi(w)$). A node is trivially reachable from itself, so $\varphi(w)$ is reachable from $\varphi(v)$.

($\Rightarrow$) Let us assume that there is a path from v to w in $FG(\varrho)$; then it must be proven that there exist v' and w' such that $\varphi(v') = v$, $\varphi(w') = w$ and $v' \Rightarrow_\varrho w'$.

Since every node in $FG(\varrho)$ represents a clique in $SG(\varrho)$, this means that there is a list of cliques $C_1, C_2 \ldots C_k$, such that there is a $y_1 \in C_1, x_2, y_2 \in C_2, \ldots, x_k \in C_k$ such that $y_1 \to x_2, y_2 \to x_3, \ldots, y_{k-1} \to x_k$.

Since every x_n and y_n are part of a clique, for every n, it is either the case that $x_n = y_n$ or $x_n \to y_n$. In both cases it is easy to construct a path that connects v' and w' from the edges mentioned in the previous paragraph.

The correspondence between dominators in the strong dependency graph and dominators in the flow graphs can now be established.

Definition 3.18 (Dominators in a control flow graph [LT79])

In a directed graph G with a distinguished node start, a node p dominates a node q if and only if every path from start to q passes through p.

Theorem 3.19

Given a repository ϱ, $v \succeq_{Is} w$ if and only if $\varphi(v)$ dominates $\varphi(w)$ in $FG(\varrho)$.

Proof

(⇐) Let us assume that every path from $start$ to $\varphi(w)$ in $FG(\varrho)$ passes through $\varphi(v)$.

1. Since every node is reachable from $start$, there must be at least one path from $start$ to $\varphi(w)$, which by the hypothesis passes through $\varphi(v)$. This means that there is a path from $\varphi(v)$ to $\varphi(w)$, and, by lemma 3.17, $v \Rightarrow w$.

2. Let x be a package in $Is(w, \varrho) \setminus Scons(v)$. Then, by definition, there is a path $x \twoheadrightarrow w$, but no path $v \twoheadrightarrow x$ in $SG(\varrho)$, and the same holds (*mutatis mutandis*) in $FG(\varrho)$. By lemma 3.16, there must exist a path $start \twoheadrightarrow \varphi(x)$ in $FG(\varrho)$.

   Given that $start \twoheadrightarrow \varphi(x) \twoheadrightarrow \varphi(w)$ is a path in $FG(\varrho)$, and v dominates w by hypothesis, this path must contain $\varphi(v)$. Since there is no path $\varphi(v) \twoheadrightarrow \varphi(x)$, $\varphi(v)$ must be on the path $\varphi(x) \twoheadrightarrow \varphi(w)$. Hence, there is a path from $\varphi(x)$ to $\varphi(v)$, and this means that $x \in Is(v, \varrho)$.

   From these two points, it follows that $v \succcurlyeq_{Is} w$.

(⇒) Let us assume that $v \Rightarrow w$ and $Is(v, \varrho) \supseteq (Is(w, \varrho) \setminus Scons(v))$.

Take any path α from $start$ to $\varphi(w)$ in $FG(\varrho)$. Observe first that, since $v \Rightarrow w$, the set of vertices $\{\varphi(x) \in \alpha \mid x \in Scons(v)\}$ is not empty (it contains at least $\varphi(w)$).

If the images of all vertices in α are in $Scons(v)$, then α is necessarily of the form $start \rightarrow \varphi(v) \twoheadrightarrow \varphi(w)$ in $FG(\varrho)$ (all other nodes in α necessarily are strong dependencies of v, so $start$ can point only to $\varphi(v)$), and the proof is complete.

Otherwise, consider the vertex $\varphi(x) \in \alpha$ which is the last one (counting from $start$) so that x is not in $Scons(v)$: since it is on a path leading to w, $x \in Is(w, \varrho)$, and since $x \notin Scons(v)$, by hypothesis we have $x \in Is(v, \varrho)$, so there is a path $\varphi(x) \rightarrow \varphi(v)$.

Consider now the vertex $\varphi(x')$ which immediately follows $\varphi(x)$ in α (see figure 3.4): by the definition of x, $x' \in Scons(v)$, so either $x' = v$ or there is a path $\varphi(v) \twoheadrightarrow \varphi(x')$. Now, if $x' \neq v$, we would have the vertices x, x', v with $\varphi(x) \rightarrow \varphi(x')$, $\varphi(x) \twoheadrightarrow \varphi(v)$ and $\varphi(v) \twoheadrightarrow \varphi(x')$ in $FG(\varrho)$; this is not possible because $FG(\varrho)$ is detransitivised. So, by necessity, $x' = v$; $\varphi(v)$ belongs to α and the proof is also complete.

We note here that the presence of cycles in $SG(\varrho)$, which are removed in $FG(\varrho)$, entails a difference in the resulting order structure.

Observation 3.20
The (flow graph) dominance relation establishes a partial order on the vertices of $FG(\varrho)$, which is an acyclic graph, while (dependency) dominance only gives a partial pre-order on $SG(\varrho)$, which may contain cycles.
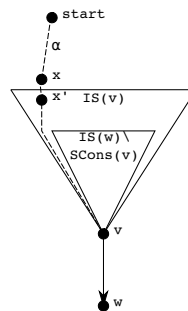

Figure 3.4: Path from start to w in dominator graph

3.4 Strong conflicts

The same reasoning that led to the introduction of strong dependencies to obtain from a repository all relevant information that is not easily available when looking only at the syntactic dependency relation can be reused, *mutatis mutandis*, for conflicts.

It is very well possible for two packages that do not have a syntactic conflict to be non co-installable; for example, if they depend on two packages that have a syntactic conflict.

A very simple example can be seen in figure 3.5. Obviously, the packages `bravo` and `charlie` are not co-installable, because there is a syntactic conflict between them.

This conflict also prevents `alpha` and `delta` from being installed together: since `alpha` depends on `bravo` and `delta` on `charlie`, installing `alpha` and `delta` will also invoke the conflict between `bravo` and `charlie`.
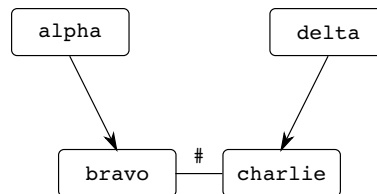
Figure 3.5: Simple strong conflict example

A slightly more complex example follows in figure 3.6. Here, none of `echo`, `alpha` and `bravo` are co-installable with neither `delta` nor `charlie`. However, since `golf` does not necessarily install `delta` (it can use `foxtrot` instead), there is no strong conflict involving `golf`.

This leads us to the following definition:

> **Definition 3.21 (Strong conflict)**
> *Given a repository ϱ, two packages p and q strongly conflict if p and q are both separately installable in ϱ, but not co-installable.*
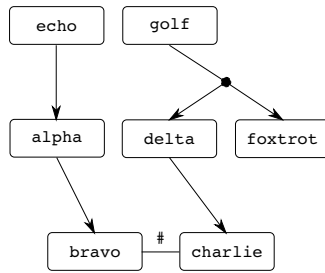
Figure 3.6: More complex strong conflict example

Similar to strong dependencies, for two packages to strongly conflict, they must be installable separately—otherwise, any non-installable package would trivially have a strong conflict with any other package.

One application of the strong conflict relationship is to find packages that have many strong conflicts: obviously, a good software distribution should try to avoid packages whose installation prevents the installation of a large set of other packages.

The set of packages whose installation is prevented by a package p is called *exclusion set* of p:

Definition 3.22 (Exclusion set)
The exclusion set *of a package p (from a repository (R, D, C)) is the set of packages $\{q \in R \mid q$ strongly conflicts with $p\}$.*

We now present some lemmas on strong conflicts that will help us define an efficient algorithm for the computation of the strong conflicts of a distribution (see chapter 4).

Lemma 3.23
Given a repository (R, D, C) and two peaceful sets $I, I' \subseteq R$, if the union $I \cup I'$ is not peaceful, there exists a conflict (c_1, c_2) such that $c_1 \in I$ and $c_2 \in I'$.

Proof $I \cup I'$ is not peaceful, so there is a conflict (c_1, c_2) with $c_1, c_2 \in I \cup I'$. Since I is peaceful, it cannot be the case that both c_1 and c_2 are in I; similarly, because I' is peaceful, it cannot be the case that both c_1 and c_2 are in I. The only possibilities are that $c_1 \in I$ and $c_2 \in I'$, or that $c_1 \in I'$ and $c_2 \in I$. In fact, because the conflict relation is symmetrical, both these cases are equivalent.

Lemma 3.24
Given a repository ϱ, if $p' \in \Delta(p)$, then there is a path from p to p' in the dependency graph of ϱ.

Proof $\Delta(p)$ is closed under the repeated application of the direct dependency function, so there must be a list $p_1, p_2, \ldots, p_n$, such that $p \to p_1$, $p_1 \to p_2$, $\ldots$, $p_n \to p'$. Therefore, $p \to p_1 \to p_2 \to \ldots \to p_n \to p'$ is a path in the dependency graph of ϱ.

These two lemmas can be used to prove a theorem about the origin of strong conflicts:

Theorem 3.25

If two packages p and q strongly conflict, there must be an explicit conflict (c_1, c_2) such that $p \twoheadrightarrow c_1$ and $q \twoheadrightarrow c_2$.

Proof Since p and q are separately installable in (R, D, C), by proposition 2.17, they also are installable in $\Delta_R(p)$ and $\Delta_R(q)$ respectively. Thus, there must be healthy sets $I_p \subseteq \Delta_R(p)$ and $I_q \subseteq \Delta_R(q)$, such that $p \in I_p$ and $q \in I_q$.

However, p and q are not co-installable, so $I_p \cup I_q$ cannot be healthy. Hence, $I_p \cup I_q$ must either be not abundant or not peaceful. Since per corollary 2.5, the union of two abundant sets is abundant, $I_p \cup I_q$ is necessarily not peaceful.

Then, per lemma 3.23, there exists a conflict (c_1, c_2) such that $c_1 \in I_p$ and $c_2 \in I_q$. Since $I_p \subseteq \Delta_R(p)$, by lemma 3.24, there is a dependency path from p to c_1; similarly, since $I_q \subseteq \Delta_R(q)$, there is a dependency path from q to c_2.

We shall use this theorem in chapter 4 to propose an efficient algorithm to compute all strong conflicts in a distribution.

Aside from making an efficient algorithm possible, there is another advantage to be drawn from this theorem: the conflict (c_1, c_2) can be seen as an *explanation* for the strong conflict between p and q—in at least one instance, the conflict (c_1, c_2) causes p and q not to be installable together.

This can be exploited by grouping the strong conflicts in a distribution by 'root cause'. An example should make things clearer:

```
2362 ppmtofb-0.32-0.1:
2362 (python-2.5.2-3 <-> ppmtofb-0.32-0.1)
* atomix-2.14.0-1 (conjunctive)
- conflict: python-2.5.2-3 - ppmtofb-0.32-0.1
- dependency: gconf2-2.22.0-1 -> python-2.5.2-3
- dependency: libgnome2-common-2.20.1.1-1 -> gconf2-2.22.0-1
- dependency: libgnome2-0-2.20.1.1-1 -> libgnome2-common-2.20.1.1-1
- dependency: atomix-2.14.0-1 -> libgnome2-0-2.20.1.1-1
(...)
```

This is the actual output of the tool we have written that generates the list of strong conflicts of a distribution. The first line tells us that the package `ppmtofb-0.32-0.1` has an exclusion set of 2362 packages; in other words, that it is not co-installable with about 10 percent of the distribution.

The second line tells us that all these 2362 strong conflicts have the syntactic conflict between `python-2.5.2-3` and `ppmtofb-0.32.0.1` as their root cause. After this second line we can see the first of the 2362 packages that have a strong conflict with `ppmtofb`, to wit `atomix-2.14.0-1`.

The dependency path between `atomix` and `ppmtofb` is shown after that; the addition (`conjunctive`) tells us that this dependency path contains only conjunctive dependencies.

This information can help us locate the problem fairly quickly. Apparently, there is a conflict between `python` and `ppmtofb` which causes this large exclusion set. This conflict must be in the metadata of either `python` or `ppmtofb`.

When we look at the actual metadata, we find that `ppmtofb` conflicts with every version of `python` that is superior to `2.4`. Since the versions of `python` currently included in Debian are all superior or equal to `2.5`, this means that `ppmtofb` cannot be co-installed with `python` (or any package that needs `python`). This explains the large exclusion set.

3.4.1 Triangle conflicts

Considering figure 3.7, we notice that the conflict between `bravo` and `charlie` is a particular one.
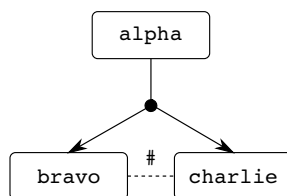
Figure 3.7: Example of a triangle conflict

If `bravo` and `charlie` have no other predecessors than `alpha`, this conflict that cannot engender any strong conflicts. Informally, the idea behind this theorem is as follows: suppose that there are two packages, `delta` and `echo`, of which one depends on `bravo` and the other on `charlie`. However, since both `bravo` and `charlie` have no other predecessors than `alpha`, both `delta` and `echo` must depend on `alpha`, and hence on both `bravo` and `charlie`. This means that `delta` and `echo` are co-installable, because it is not necessary to install both `bravo` and `charlie` to satisfy all dependencies.

This notion, which will be proven in detail in the remainder of this section, allows us to optimise the algorithm to compute the strong dependencies in a distribution, because all triangle conflicts can be discounted. Practical experimentation shows that there are not very many triangle conflicts, but they show up often. As an example, in Debian, the `debconf` package, which is depended on by a large proportion of the distribution, is the 'apex' of a triangle conflict with `debconf-english` and `debconf-i18n`. Discounting this conflict considerably reduces the search space (for more information, see chapter 4).

Definition 3.26 (Triangle conflict)
A conflict (c_1, c_2) is a triangle conflict *if and only if there exists a package p such that:*

- *there is a $d \in D(p)$ such that $\{c_1, c_2\} \subseteq d$;*
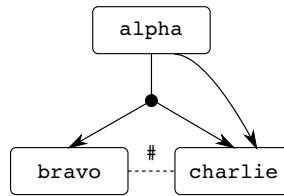

Figure 3.8: Example of a degenerate triangle conflict

> • *there is no other p' such that $p' \rightarrow c_1$ or $p' \rightarrow c_2$.*

This definition allows for a "degenerate" triangle conflict, as shown in figure 3.8. In this figure, the conjunctive dependency from `alpha` to `charlie` obsoletes the disjunctive dependency from `alpha` to `bravo` and `charlie`; since `charlie` is always going to be installed, the disjunctive dependency is always satisfied and hence superfluous (at least with respect to installability).

Such degenerate triangle conflicts can hinder in the proof, since the assumption is that in a triangle conflict, one can choose either side of the conflict to satisfy the dependency—something that is obviously not true in case of a degenerate triangle conflict.

> **Definition 3.27 (Removal of superfluous dependencies)**
> *Given a repository (R, D, C), for all $p \in R$, remove all dependencies $d \in D(p)$ that satisfy the following condition:*
>
> $$\exists_{d' \in D(p)}[d \subset d']$$

Observe that doing this does not hinder installability: no new conflicts are introduced, so a peaceful distribution remains peaceful; furthermore, any alternative d that is removed has a strict subset d' that is not removed— and any package that satisfies d' also satisfies d.

Now the proof of our main theorem can begin, which notes that packages that contain only triangle conflicts in their dependency cones are always co-installable:

> **Theorem 3.28**
> *Given:*
>
> • *A repository $\varrho = (R, D, C)$, from which superfluous dependencies have been removed as per definition 3.27;*
>
> • *A set of packages $a_1, a_2, \ldots, a_n \in R$, such that there does there does not exist any c and i ($1 \leq i \leq n$) such that $(c, a_i) \in C$;*

> - *For every i such that $1 \leq i \leq n$, a set $A_i \subseteq R$ such that A_i is healthy and $a_i \in A_i$ (in other words, a_i is installable in R, using A_i as the installation set);*
>
> - *For all packages $c_1, c_2 \in \bigcup_{i=1}^{n} \Delta(a_i)$ with $(c_1, c_2) \in C$, (c_1, c_2) is a triangle conflict.*
>
> *Then $a_1, a_2, \ldots, a_n$ are co-installable with respect to ϱ.*

Proof Let us define a function $K(A, B)$, where A and B are sets of packages, as a list of packages from A that are involved in a conflict spanning A and B;
$K(A, B) = \{p \in A \mid \exists_{q \in B}[(p, q) \in C]\}$
Furthermore, let us define a function α_k such that:

- $\alpha_1 = A_1$

- $\alpha_k(1 < k \leq n) = (\alpha_{k-1} \cup A_k) \setminus K(\alpha_{k-1}, A_k)$

It is easy to see that since $a_i \in A_i$, $\{a_1, a_2, \ldots, a_n\} \subseteq \alpha_n$.
Now, α_n is abundant. Proof by induction:

- A_1 is abundant, so α_1 is abundant.

- Supposing that α_{k-1} is abundant, then by corollary 2.5, $\alpha k - 1 \cup A_k$ is abundant. Therefore, for any package $p \in \alpha_{k-1} \cup A_k$ and any $d \in D(p)$, there must be an $x \in d \cap (\alpha_{k-1} \cup A_k)$. Now, either $x \in K(\alpha_{k-1}, A_k)$, or not. In the first case, there is a conflict (x, y), with both x and y in $\alpha_{k-1} \cup A_k$. Now, since $x \in K(\alpha_{k-1}, A_k)$, it is not in α_k, but d can still be satisfied by using y (since p, x and y form a triangle conflict). Hence, α_k is abundant. In the second case, trivially, $x \in \alpha_k$.

α_n is also peaceful. Proof by induction:

- A_1 is peaceful, so α_1 is peaceful.

- Supposing that α_{k-1} is peaceful, then by lemma 3.23, $\alpha_{k-1} \cup A_k$ is peaceful if there are no conflicts (c_1, c_2) with $c_1 \in \alpha_{k-1}$ and $c_2 \in A_k$. When we look at the construction of α_k, we see that for any such conflict, one of its packages from $\alpha_{k-1} \cup A_k$ is removed, so that α_k is indeed peaceful.

Algorithms 4

In the previous chapters, we have presented a model of F/OSS distributions and their properties. Such a model is useful as a basis for reasoning about the properties of distributions (as we have already done in chapter 3), but it can also be used as a starting point for distribution analysis.

Given the scale of distributions, such analyses must necessarily be highly automated. Furthermore, their implementation must be efficient, since the analyses will be run on large distributions, preferably on a daily basis in order to follow the evolution of distributions over time.

Therefore, in this chapter, we shall explain the algorithms used to implement the notions presented in chapters 2 and 3. Frequent reference will be made to these chapters, as the theorems proposed there are used for optimisation of the algorithms.

We shall also discuss the theoretical complexity of the algorithms.

In chapter 7, we shall discuss the practical applicability of these algorithms (for example in terms of running time), and talk in more detail about some more interesting results found in their output.

4.1   Installability

As seen in definition 2.8, for a package to be installable, all its dependencies (and their dependencies, and the dependencies of those dependencies, etc.) must be satisfied (i.e. the install set must be abundant, cf. definition 2.4), and there must not be any conflicts between the packages used to satisfy the dependencies (i.e. the install set must be peaceful, cf. definition 2.6).

An efficient way to check these conditions is by making use of a SAT solver; the installability problem translates easily into a SAT specification, as we shall show now.

In order to translate a dependency $D(p) = \{\{x_1^1, x_1^2, \ldots, x_1^{n_1}\}, \{x_2^1, \ldots\}, \ldots\}$ into SAT, the following clauses can be used:

$$\neg p \vee x_1^1 \vee x_1^2 \vee \ldots \vee x_1^{n_1}$$
$$\neg p \vee x_2^1 \vee x_2^2 \vee \ldots \vee x_2^{n_2}$$
$$\vdots$$

A conflict (p, q) is specified as follows:

$$\neg p \vee \neg q$$

Note that it is perfectly possible to satisfy all these clauses by simply assuming $\neg p$. This is quite natural: installing no packages at all means that there are no conflicts. Only when one actually specifies a package that must necessarily be installed can a problem occur.

Let us look at an example: the repository from figure 2.1. Its SAT encoding would look like this:

```
¬alpha ∨ bravo
¬alpha ∨ charlie ∨ delta
¬echo ∨ delta ∨ foxtrot
¬delta ∨ ¬foxtrot
```

In order to solve the installability problem for the package `alpha`, for example, it suffices to add the single clause `alpha` to the SAT encoding (to force the variable `alpha` to be set to `true`), and then run a SAT solver.

Conversely, any SAT problem can also be translated into into a package installation problem. Given a SAT problem of n clauses $C_1, C_2, \ldots, C_n$, such that $C_i = c_1^i \vee c_2^i \vee \ldots \vee c_{m_i}^i$, where any c_j^i is either a variable v or the negation of a variable $\overline{v}$, where $\{v_1, v_2, \ldots, v_k\}$ is the set of possible variables.

Then, let us define the following repository (R, D, C):

$$R = \{P, P_{C_1}, P_{C_2}, \ldots, P_{C_n}, P_{v_1}, P_{v_2}, \ldots, P_{v_k}, P_{\overline{v_1}}, P_{\overline{v_2}}, \ldots, P_{\overline{v_k}}\}$$
$$D(P) = \{\{P_{C_1}\}, \{P_{C_2}\}, \ldots, \{P_{C_n}\}, \{P_{v_1}, P_{\overline{v_1}}\}, \{P_{v_2}, P_{\overline{v_2}}\}, \ldots, \{P_{v_k}, P_{\overline{v_k}}\}\}$$
$$D(P_{C_1}) = \{\{P_{c_1^1}, P_{c_2^1}, \ldots, P_{c_{m_1}^1}\}\}$$
$$\vdots$$
$$C = \{(P_{v_1}, P_{\overline{v_1}}), (P_{\overline{v_1}}, P_{v_1}), \ldots\}$$

Now, our SAT problem is solvable if and only if the package P is installable. This brings us to the following proposition:

> **Proposition 4.1**
> *The package installability problem is NP-hard.*

Proof As seen above, a package installability problem can be translated (in polynomial time) into a SAT problem and vice versa. Thus, the package installability problem is equivalent in complexity to the SAT problem, which is NP-hard.

In practice, luckily, the installability problems remain tractable, since the number of conflicts and alternatives are limited with respect to the main distribution (see chapter 7 for more information).

In the MANCOOSI project, much work has been done to optimise SAT solvers for package installation problems.

4.2 Strong dependencies

The algorithm to check whether there is a strong dependency between two packages p and q (that is, $P \Rightarrow Q$) is a slight variation on the SAT encoding

described in the previous section. It consists in checking the satisfiability of the following formula:

$p \land \neg q \land SAT(R)$

In this formula, $SAT(R)$ denotes the formula obtained by the translation into SAT clauses of the dependencies and conflicts of the repository R.

The SAT problem shown above is solvable if and only if it is possible to install p in R without also installing q. If this is not the case, by necessity p strongly depends on q.

Note that this problem is the complement of an NP-complete problem (the crux is to find out whether a SAT formula is *not* solvable). This means that the problem, theoretically, is exponential in complexity (it is part of the co-NP-hard class).

In practice, like the installability problem, the problems remain tractable; at least if one just wants to check whether a package strongly depends on another. However, if one wants to draw the strong dependency graph of an entire distribution, using this algorithm for every pair of packages in a distribution R would entail doing $O(|R|^2)$ SAT checks. With distributions numbering in the tens of thousands of packages, this is not feasible in any practical time.

However, using corollary 3.7, it becomes possible to dramatically reduce the number of SAT checks: because any installation set is a superset of the set of strong dependencies of a package, it suffices to find an install set of that package, and then we only need to check the members of this install set. This idea is implemented in algorithm 3.

Algorithm 3 Computation of strong dependencies, version 1

---

for $p \in R$ **do**
  $S(p) \leftarrow \emptyset$
  $I \leftarrow \texttt{install}(p)$
  for $i \in I$ **do**
    if $\texttt{strongdep}(p, i)$ **then**
      $S(p) \leftarrow S(p) \cup i$
    end if
  end for
end for
return S

This is especially efficient if we use an algorithm that finds minimal install sets, because then the number of invocations of the SAT solver remains as small as possible.

The algorithm can be optimised further by using corollary 3.2. One can compute the conjunctive dependencies of p very quickly (by traversing the syntactic dependency graph); since by corollary 3.2, these are strong dependencies as well, they do not need to be checked by the SAT solver. The only dependencies that remain to be checked are the members of the install set that are not conjunctive dependencies.

The complete algorithm that uses this corollary is algorithm 4.

Let us present some examples of how this works. Consider the example distribution from figure 4.1:

Dotting the arrows that are not strong dependencies produces the graph as

---

Algorithm 4 Computation of strong dependencies, version 2

---

for $p \in R$ **do**
   $S(p) \leftarrow \emptyset$
   $I \leftarrow \texttt{install}(p)$
   for $i \in I$ **do**
     if $\texttt{conjunctive\_dep}(p, i)$ **then**
       $S(p) \leftarrow S(p) \cup i$
     else if $\texttt{strongdep}(p, i)$ **then**
       $S(p) \leftarrow S(p) \cup i$
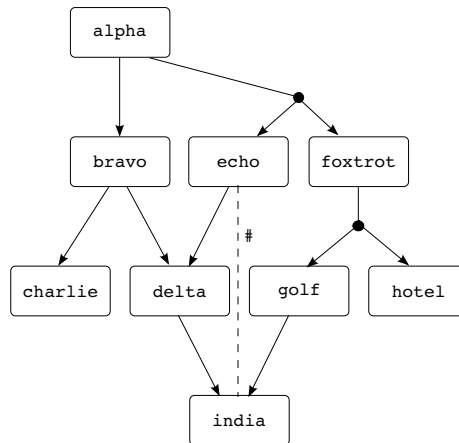   **end if**
   end for
end for
return S

Figure 4.1: Example distribution
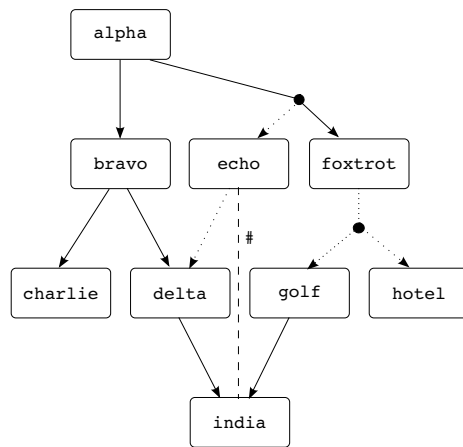
Figure 4.2: Example distribution with strong dependencies
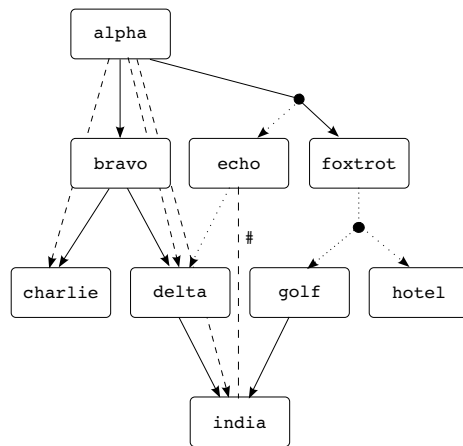
Figure 4.3: Example distribution with transitive strong dependencies

shown in figure 4.2. Note that `delta` is *not* a strong dependency of `echo`, even though they are conjunctive dependencies: because `echo` depends on `hotel`, but also conflicts with it, `echo` is not installable, and therefore it does not have any strong dependencies.

And then finally, after adding the transitive strong dependencies (as dashed lines), we get figure 4.3.

In order to generate the transitive graph, the transitive edges can be added on the fly during the computation, using the algorithm suggested in [PvL88] (this algorithm actually computes both the transitive reduction and the transitive closure, but we have removed the transitive reduction part). This is algorithm 5.

The `add_edge` algorithm adds an edge (v, v') to the graph, plus any transitive edges that might be needed. This works as shown in algorithm 6.

Using these optimisations, the algorithm runs quite quickly, even on large

---

Algorithm 5 Computation of transitive strong dependency graph

Require:

- $S = (V_S, E_S)$ is the syntactic dependency graph

- $f_S : E_s \rightarrow \{\texttt{Conjunctive}, \texttt{Disjunctive}\}$ is its annotation function.

$V \leftarrow V_S$
$E \leftarrow \emptyset$
for $v \in V_S$ **do**
  for $v' \in \text{succ}_S(v)$ **do**
    if $f_S(v, v') = \texttt{Conjunctive}$ **then**
      $\texttt{add\_edge}(v, v')$
    else if $\texttt{strongdep}(v, v')$ **then**
      $\texttt{add\_edge}(v, v')$
    end if
  end for
end for
return (V, E)

Algorithm 6 Adding an edge to a transitive graph

Require:

- V, E is a transitive graph that does not contain any edges (p, p)

- (i, j) is an edge to be added to this graph

$E \rightarrow E \cup (i, j)$
mark j red
for $k \in i \cup \text{pred}(i)$ **do**
  if $(k, j) \notin E$ **then**
    mark j red
    while there are red nodes **do**
      let l be a red node
      unmark l
      if $k \neq l$ **then**
        $E \rightarrow E \cup (k, l)$
        for $m \in \text{succ}(l)$ **do**
          if $(k, m) \notin E$ **then**
            mark m red
          end if
        end for
      end if
    end while
  end if
end for

---

distribution. More information about this can be found in chapter 7.

4.3 Dominators

The classic dominator algorithm exactly follows the definition of strong dominators (definition 3.8); it looks at the successors of all vertexes and checks if there is a dominance relationship. If this is the case, it adds the appropriate edge to the graph. This classic algorithm is presented as algorithm 7.

Algorithm 7 Classic algorithm for dominance

Require: R is a repository
Require: (V, E) is the (transitive) strong dependency graph for R
  $V_D \leftarrow V$
  $E_D \leftarrow \emptyset$
  for $p \in V$ **do**
    for $q \in \text{succ}(p)$ **do**
      if $Is(p, R) \supseteq Is(q, R) \setminus Scons(p)$ **then**
        $E_D \leftarrow E_D \cup (p, q)$
      end if
    end for
  end for

This algorithm can be adjusted for relative dominators; it suffices to slightly adjust the central test, as shown in algorithm 8:

Algorithm 8 Classic algorithm for relative dominance

Require: R is a repository
Require: (V, E) is the (transitive) strong dependency graph for R
Require: f is the fraction of relative dominance allowed
  $V_D \leftarrow V$
  $E_D \leftarrow \emptyset$
  for $p \in V$ **do**
    for $q \in \text{succ}(p)$ **do**
      $fv \leftarrow \frac{|(Is(q,R) \setminus Scons(p)) \setminus Is(p,R)|}{|Is(p,R)|}$
      if $fv \leq f$ **then**
        $E_D \leftarrow E_D \cup (p, q)$
      end if
    end for
  end for

We have shown in theorem 3.19 that the notion of dominance in flow graphs is equivalent to the notion of dominance proposed in this thesis. There is an algorithm of complexity $O(|V|+|E|+|E|\log|V|)$ ([LT79]) for finding dominators in flow graphs.

This algorithm only works on a non-transitive graph, however, and the theoretical complexity of computing the transitive reduction of a graph is $O(|V|^3)$. We will see in chapter 7 that due to the specific characteristics of the strong dependency graph, the transitive reduction in actual cases can be done very

quickly. In consequence, the Tarjan algorithm is much faster than the 'standard' algorithm presented above.

Another disadvantage of the Tarjan algorithm is that it cannot be used to compute relative dominance graphs.

The Tarjan algorithm is algorithm 9.

Algorithm 9 Fast Tarjan algorithm for dominance

Require: R is a repository
Require: (V, E) is the strong dependency graph for R
  $(V, E) \leftarrow \text{transitive\_reduction}((V, E))$
  $(V, E) \leftarrow \text{cycle\_reduction}((V, E))$
  $V \leftarrow V \cup \{\texttt{start}\}$
  for $v \in V$ **do**
    if $\text{pred}(v) = \emptyset$ **then**
      $E \leftarrow E \cup \{(\texttt{start}, v)\}$
    end if
  end for
  $\text{lengauer\_tarjan}(V, E)$

4.4 Strong conflicts

Similarly to strong dependencies, checking whether two packages p and q strongly conflict can easily be done with a SAT check:

    $p \wedge q \wedge SAT(R)$

If there is no solution for this formula, p and q strongly conflict (and otherwise they do not).

Like for the strong dependency problem, this problem is the complement of a SAT problem; therefore it is in the co-NP-complete class and potentially exponential in complexity. Again, however, if one were to use this to compute every strong conflict for an entire distribution, this would result in our doing $O(|R|^2)$ SAT checks.

To avoid this, we propose an algorithm that uses theorem 3.25. This theorem indicates that if packages p and q strongly conflict, there must be a conflict (c_1, c_2) such that there is a dependency path from p to c_1 and from q to c_2.

This means that the search space can be reduced drastically, because all pairs of packages for which the aforementioned condition does not hold can be safely ignored.

The proposed implementation turns this around by starting from the explicit conflicts (c_1, c_2) in a distribution and then looking at all the elements of $\overline{\Delta}(c_1) \times \overline{\Delta}(c_2)$[1]. By theorem 3.25, every such element could be a pair of packages that strongly conflict with each other.

Even though the number of explicit conflicts is normally limited (in the latest Debian stable distribution, for example, which contains some 22 000

---

[1]See definition 2.16.

packages, there are only about 1 000 explicit conflicts), their reverse dependency cones can be quite large; therefore, even though the search space is reduced by about two thirds, the number of candidates remains significant.

Consider a candidate (p, q). Here, p and q are packages, and there exists a conflict (c_1, c_2) such that $p \twoheadrightarrow c_1$ and $q \twoheadrightarrow c_2$. This in itself does not guarantee that p and q strongly conflict; but if $p \overset{c}{\twoheadrightarrow} c_1$ and $q \overset{c}{\twoheadrightarrow} c_2$, then any installation of p and q would necessarily include both c_1 and c_2 and hence not be peaceful.

This means that every candidate that is connected to its root conflict solely by *conjunctive* dependencies is automatically a strong conflict and thus does not require a SAT check. When we look at the results for Debian stable, as an example, 80 percent of the strong conflicts found satisfy this condition.

The search space can be reduced even further by using theorem 3.28. This theorem shows that packages that have only triangle conflicts in their dependency cones are co-installable. It follows that triangle conflicts can be discounted for the generation of strong conflict candidates: for two packages to be in strong conflict, there must be at least one non-triangle conflict in one of their dependency cones. Discounting triangle conflicts will therefore not change our result.

We will see in chapter 7 that these measures speed up the algorithm considerably: even though there are very few triangle conflicts, some of the triangle conflicts that are there have very large reverse dependency cones.

The complete algorithm is shown as algorithm 10.

The algorithm to find conjunctive dependencies is in fact slightly more optimised than shown in the figure above: instead of computing the reverse dependency cone and then checking if there is a conjunctive path, these two operations are combined: while constructing the reverse dependency cone, the algorithm keeps track of which elements of the cone have a conjunctive dependency path and which do not; elements which have a conjunctive dependency path are immediately added to the list of strong dependencies, whereas other elements are added to the list of pairs that require a SAT check.

Here is the algorithm that constructs the reverse dependency cone in this way. There are two arguments: P the set of packages of which the cone has to be computed, and V the set of packages that have already been visited (initially empty). The algorithm, algorithm 11, returns a pair of sets: first the conjunctive predecessors, then the disjunctive predecessors.

The idea here is that the algorithm continually takes any conjunctive predecessor of a package that it has not yet visited. This will result in a set C of all predecessors that have conjunctive dependency paths; after this, the algorithm takes the reverse dependency cone of C to get all disjunctive predecessors as well. Since the dependency graph may contain cycles, we must subtract C from the set of disjunctive predecessors to avoid duplicates.

Let us note in passing that this algorithm terminates; packages are never visited twice, so since the repository is finite, at some point there are either no more predecessors to be found, or every package in the repository has been visited.

Algorithm 10 Computation of strong conflicts

Require: (R, D, C) is a distribution
  remove superfluous dependencies {see definition 3.27 in chapter 3}
  $S \leftarrow \emptyset$ {set of strong conflicts}
  $P \leftarrow \emptyset$ {set of *possible* strong conflicts}
  for $(c_1, c_2) \in C$ **do**
    if not triangle(c_1, c_2) **then**
      for $p_1 \in \overline{\Delta}_R(c_1)$ **do**
        for $p_2 \in \overline{\Delta}_R(c_2)$ **do**
          if $p_1 \overset{c}{\twoheadrightarrow} c_1$ **and** $p_2 \overset{c}{\twoheadrightarrow} c_2$ **then**
            $S \leftarrow S \cup (p_1, p_2)$
          else
            if not $(p_2, p_1) \in C$ **then** {strong dependencies are symmetric}
              $P \leftarrow P \cup (p_1, p_2)$
            end if
          end if
        end for
      end for
    end if
  end for
  for $(p_1, p_2) \in P$ **do**
    if co-installable(p_1, p_2) **then**
      $S \leftarrow S \cup (p_1, p_2)$
    end if
  end for
  return S

Algorithm 11 Computation of the dependency cone

  $C \leftarrow \emptyset$ {conjunctive predecessors}
  while $P \neq \emptyset$ **do**
    take a p from P
    if $p \notin V$ **then**
      $C \leftarrow C \cup \{x \in \text{pred}(p) \mid x \overset{c}{\to} p\}$
      $V \leftarrow V \cup \{p\}$
      $P \leftarrow P \cup C$
    end if
  end while
  return $(C, \overline{\Delta}_R(C) \setminus C)$

Tools 5

In this chapter, we shall give an overview of the different tools that have been developed over the course of the EDOS and MANCOOSI projects.

5.1 distcheck

This tool, developed by Jerôme Vouillon at the very start of the EDOS project, is used to check for non-installable packages in a distribution. The SAT solver at its heart (a re-implementation of Mini-SAT) is used for determining package (co-)installability in every tool mentioned hereafter.

Simply checking for non-installable packages is worthwhile in itself, but distcheck does more: it also provides an explanation as to *why* a package is not installable. As we have seen in definition 2.8 from chapter 2, for a package to be installable there has to be an abundant and peaceful subset of the repository that contains it. The possible reasons for a package not being installable are thus exactly two: there not being an abundant set (i.e. there is a dependency that cannot be satisfied), or there not being a peaceful set (i.e. a package depends on two conflicting packages, and this conflict cannot be avoided). For example:

```
python-gnuradio (= 3.0.4-2): FAILED
The following constraints cannot be satisfied:
python-gnuradio (= 3.0.4-2) depends on python (« 2.5) {NOT AVAILABLE}

python-wxgtk2.4 (= 2.4.5.1.1+b1): FAILED
The following constraints cannot be satisfied:
python-wxgtk2.4 (= 2.4.5.1.1+b1) depends on python-wxversion
{python-wxversion (= 2.6.3.2.2-2)}
python-wxgtk2.4 (= 2.4.5.1.1+b1) conflicts with python-wxversion (=
2.6.3.2.2-2)
```

The first package, `python-gnuradio`, is an example of a package that cannot be installed because of a non-satisfiable dependency; the second, `python-wxgtk2.4` cannot be installed because it simultaneously depends on and conflicts with the same package.

5.2 dose

The dose library (Distribution Object Storage Engine) is the library used by all EDOS tools. It provides an API for storage, manipulation and reporting of all

types of package distribution algorithms.

There have been two main versions of dose: version 2, written originally by Berke Durak and extended fairly heavily by myself, and version 3, written by Pietro Abate, in cooperation with myself.

The basic make-up of both versions is the same: a package store, with an API to manipulate packages and distributions, and to execute some of the algorithms. The implementation, however, is quite different.

5.2.1 dose2

The main goal in designing dose, version 2, was to have a library for storing multiple snapshots of a distribution, from different dates, in such a way that they would take a minimum of space, and be quickly accessible. The reason for these requirements was that dose was intended as a back-end for the anla web interface (described below).

The obvious way of doing this is using an SQL database. This solution was not used, however, mainly because of performance issues (the state of the OCaml libraries for SQL databases was, at that time, not up to the task of handling databases with years' worth of distributions).

Instead, a 'dosebase' consists of an index in dbm format; this index contains a record of every package, the distribution it is part of, and its lifetime, i.e. the dates on which it was present in the distribution. Furthermore, it contains pointers to separate files that contain the actual package metadata.

In order to be able to quickly respond to queries, this 'dosebase' is only used as a persistent storage; all package metadata is loaded into memory upon starting a dose2 application. This provides for great speed (once the initial loading is done, responses to even the most complicated queries are instantaneous), but obviously takes up a large amount of memory.

In practice, though, it is possible to work with dose archives containing several years' worth of distributions; memory usage in this case will be in the order of 1 Gb.

In memory, the package metadata is stored in a simple table (the same structure is used for all distribution formats; all packages in one instance of dose2 must be of the same format), without translation, except for the version numbers. In order to avoid having to call the complicated version comparison algorithm, every version number is translated into a pair of integers (one integer for the epoch and version proper, and one integer for the release). In this way, instead of having to parse the entire version number, two integer comparisons suffice.

It is not possible to use a single integer, since, as seen in chapter 2, the RPM version order is not a complete order (for example, a version `1.27` is equal to `1.27-1` and `1.27-2`, but `1.27-1` is inferior to `1.27-2`).

Dose2 supports the Debian, RPM and NetBSD pkgsrc package formats.

On top of this index is a layer that deals with retrieving packages by name, date or distribution.

Furthermore, there are functions to deal with dependencies, dependency cones, installability (the SAT solver from distcheck is integrated into dose2), strong dependencies and conflicts.

Figure 5.1 gives an overview of the structure of the dose2 library.

Taking the modules from top to bottom, the `rapids` module is the module that takes care of storage in memory, with indexes for rapid referral. The storage is filled using the `waterway` module, which uses `napkin` as a generic package data structure, and `ocamlrpm` and `ocamldeb` as package-specific parsers. The algorithms are mostly in the `packetology` module; `lifetime` deals with dates and times. And finally, the `satsolver` module is used by `packetology` to solve installability problems.

5.2.2 dose3

Dose3 is a complete rewrite of dose2. It has the same functionality, but there are several differences in implementation. When using dose2, it turned out that the functions for archiving distributions over time were used less frequently than the functions for considering just one universe of packages and running algorithms on those.

Furthermore, the CUDF format [TZ09] had been developed as a distribution-agnostic way of storing package metadata.

With this experience in mind, it was decided to reimplement dose2, using CUDF as a universal data structure. This means that when merging a distribution, it is first translated into CUDF, after which the different algorithms can be executed.

The dose2 data structures also have this property of universality with regard to the distribution format, but they are much more complicated.

For some algorithms, mostly the time-intensive ones such as the generation of the strong dependency or strong conflict graph, there is an extra translation involved: every package is given an ID and dependencies and conflicts are translated into lists of (or lists of lists of) these IDs. This allows to reduce memory usage and increase speed for these algorithms.

5.3 Ceve

Ceve (the name is supposed to have meant *something*, but exactly what has been lost in the mists of time) was written by me, as a generalised package format parser and translator.

Its first function was to translate package repositories into SAT specifications, in order to be able to compare different, external solvers, and to create a dependency graph out of a repository using the EGraph format, a derivative of the GraphML XML format.

Later on, the possibility to create databases out of repositories has been added: both SQL databases and the Dose2 format have been supported. Graphs can now also be output in the Dot format.

It is also possible to manipulate the data between parsing and output, for example to output only the dependency cone of a certain package, or to eliminate virtual packages (by replacing them in dependencies with the list of packages that provide them).

Ceve has now been rewritten using Dose3.

5.4   Pkglab

Pkglab is a front-end to the DQL (Distribution Query Language). This language is a domain-specific query language for distributions; the pkglab tool, therefore, can execute and report all sorts of queries on distributions and packages.

In order to explain the workings of the pkglab tool, we shall start with an example session. First, at the start of the program, one 'merges' a package repository, which means that the contents of that repository are loaded into our dose2 backend in memory.

```
pkglab $Revision: 5129 $ by the MANCOOSI Project
> #merge "deb:/home/users/boender/data/debian/history/20100129-debian-5.0.4-Packages"
Merging "deb:/home/users/boender/data/debian/history/20100129-debian-5.0.4-Packages"...
Completing conflicts... * 100.0%
>
```

The pkglab tool knows a number of *directives*, preceded by a hash sign. These directives are not part of the DQL, and have to do with its general operations, such as input (merging), output, and exiting.

Now that a repository has been merged, one can use a DQL function to show the list of packages known to pkglab:

```
> packages
{ zzuf'0.12-1, libzorp2-dev'3.0.8-0.5, zoph'0.7.1-1lenny1@all, zope3'3.3.1-7,
zope3-sandbox'3.3.1-7@all, ... }
```

A package name consists of a *unit* (zzuf, for example), an apostrophe, a version number (0.12-1), and optionally, an at sign and an architecture all. It is possible to have a default architecture; if the architecture of a package is equal to the default architecture, it will not be mentioned (as is the case for the zzuf'0.12-1 package above).

Obviously, the number of packages is fairly large. We can use a DQL function to compute how large exactly:

```
> count(packages)
22299
```

A simple operation is to find out whether there are any broken packages in the repository. For this, there is DQL's check function:

```
> check(packages, packages)
Conflicts and dependencies... * 100.0%
Solving * 100.0%
<diagnosis:closure size 22299, 4 failures>
```

The check function takes two arguments: the first is the set of packages to check, the second is the set of available packages. We see from the result that there are 4 packages for which the installation has failed. If one wants to obtain more information, the result of the check function must be assigned to a variable and the show directive used:

```
> $d <- check(packages, packages)
Conflicts and dependencies... * 100.0%
Solving * 100.0%
> #show $d
Diagnosis:
Conflicts: 2014
Disjunctions: 96357
Dependencies: 101920
Failures (total 4):
Package libpils-dev'2.1.3-6lenny4@all cannot be installed:
libpils-dev'2.1.3-6lenny4@all depends on one of:
- heartbeat-dev'2.1.3-6lenny4@i386
libpils-dev'2.1.3-6lenny4@all and heartbeat-dev'2.1.3-6lenny4@i386 conflict
```

The tool outputs a diagnosis for every one of the failed packages, but only one has been displayed here. The diagnosis is the same as that of `distcheck`, but displayed slightly differently.

Other functions in the same vein are `check_together`, which checks for the co-installability of packages (the `check` function checks all elements of its first argument separately, whereas the `check_together` function checks them together); and `install`, which returns an installation set instead of a diagnosis.

Note also the use of variables: the name of a variable is always prefixed with a dollar sign; and the `<-` operator is used for assignment.

There are also functions to show information about specific packages:

```
> depends(a7xpg'0.11.dfsg1-4)
[ [[. a7xpg-data (= '0.11.dfsg1-4) .]]; [[. libc6 (>= '2.7-1) .]]; [[. libgcc1 (>=
'1:4.1.1-21) .]]; [[. libgl1-mesa-glx .];[. libgl1 .]]; [[. libsdl-mixer1.2 (>= '1.2.6) .]];
[[. libsdl1.2debian (>= '1.2.10-1) .]]; [[. zlib1g .]] ]
> conflict_list(liba52-0.7.4-dev'0.7.4-11)
[[. a52dec .];[. a52dec-dev .];[. liba52-dev .]]
```

The first command shows us the dependency specification of the `a7xpg` package. There is a dependency on `a7xpg-data`, with a specific version, a dependency on `libc6`, a dependency on `libgcc1`; then, an alternative dependency on either `libgl1-mesa-glx` or `libgl1`, and so forth. The same goes for the conflict list.

The objects enclosed by `[.` and `.]` are version specifications. We can find out which packages conform to these specifications by using the `select` function:

```
> select([. a52dec .])
{ liba52-0.7.4-dev'0.7.4-11 }
> select([. mail-transport-agent .])
xmail'1.25-4, ssmtp'2.62-3, sendmail-bin'8.14.3-5, postfix'2.5.5-1.1,
nullmailer'1:1.4-1.1, msmtp-mta'1.4.15-1@all, masqmail'0.2.21-4, exim4-daemon-light'4.69-9,
exim4-daemon-heavy'4.69-9, esmtp-run'0.6.0-1@all, courier-mta'0.60.0-2, citadel-mta'7.37-8
```

The `select` command returns *all* known packages that satisfy the given version specification; if multiple repositories are loaded, and one only wants packages from a specific repository, the intersection operator can be used.

If a dose2 dosebase has been loaded, it is possible that there are multiple archives available. This can be checked using the `archives` function:

```
> archives
{ %debian/testing/main/i386, %debian/testing/contrib/i386, %debian/unstable/main/i386,
%debian/unstable/non-free/i386, %debian/stable/main/i386, %debian/stable/contrib/i386,
%debian/unstable/contrib/i386, %debian/testing/non-free/i386, %debian/stable/non-free/i386 }
> count(contents(%debian/testing/main/i386,2008-01-01))
20747
> count(contents(%debian/testing/main/i386,2008-09-15))
22638
```

Here, we can see the difference between a plain `select` and a `select` with an intersection:

```
> count(select([. mail-transport-agent .]))
91
> count(select([. mail-transport-agent .]) & contents(%debian/testing/main/i386,2008-01-01))
12
```

It is possible to use higher-order functions such as `map` and `filter`; cf. this example, which shows us the number of packages that have no direct dependencies:

```
> count(filter(packages,$a -> depends($a) = []))
9679
```

It is also possible to use regular expressions to select packages textually, such as all packages that contain the text `ocaml`:

```
> packages   /ocaml/
{ libcore-ocaml-dev'0.5.0-5, libnumerix-ocaml'0.22-4+b2, libsqlite-ocaml-dev'0.3.5.arch.4-8,
libocamlnet-ocaml'2.2.9-2+b1, libequeue-gtk2-ocaml-dev'2.2.9-3@all, ... }
```

This is but a short overview of the possibilities of the pkglab tool: using these and other functions, it is possible to express complicated queries. As an example, pkglab has been used to determine the correctness of an automatic dependency generation algorithm, where it was used to check if there was no difference in installable packages between a repository generated with the new algorithm and a normal repository [BDCV+08].
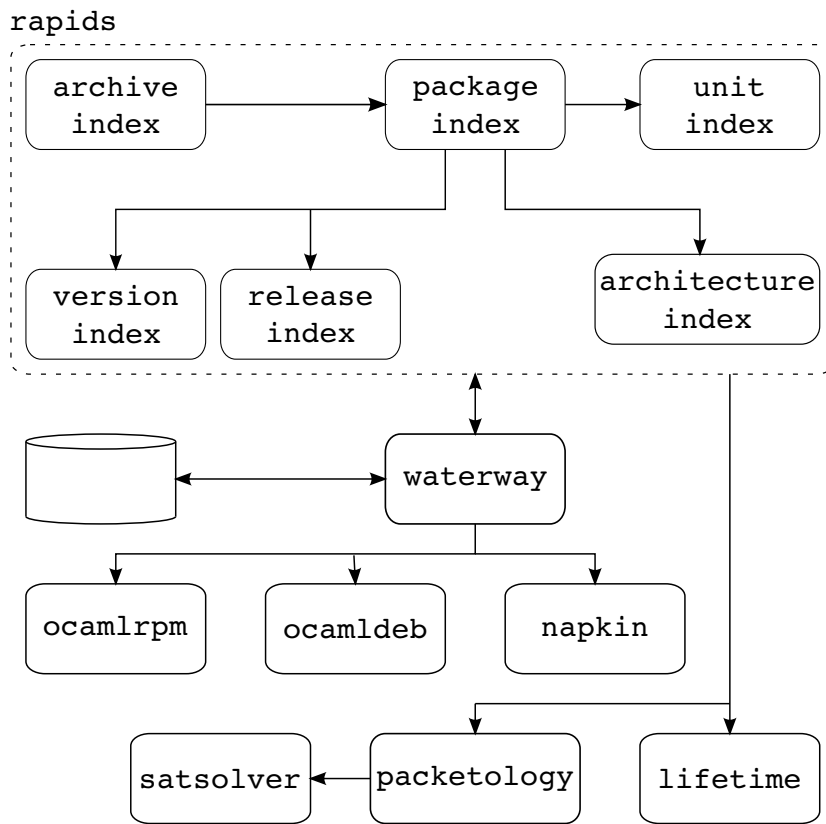
Figure 5.1: Structure of the dose2 library
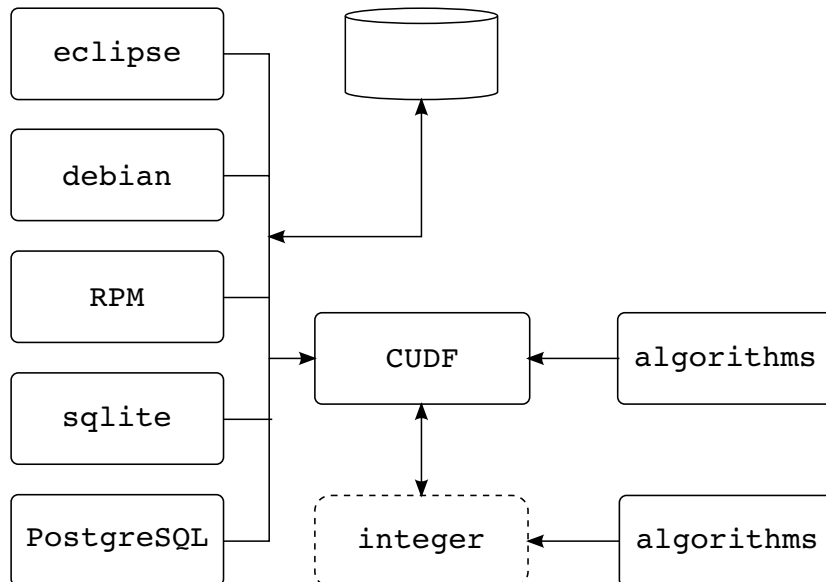
Figure 5.2: Dose3 structure

Formalisation 6

In this chapter, we shall introduce a formalisation of the theorems presented in the previous chapters, together with their proofs. For this, we have used the `Coq` proof assistant.

Obviously, such a formalisation allows us to control the correctness of our proofs, and even results in additions to the theory; as an example, the notion of a degenerate triangle conflict as shown in figure 3.6 in chapter 3 was coined when formalising the proof of theorem 3.28.

The ultimate goal of this formalisation is to be able to verify the correctness of the algorithms proposed in chapter 4. This is discussed in more detail in the 'Future work' paragraph in the conclusion.

6.1   Repository

As seen in chapter 2, a repository is a tuple that contains a set of packages, a set of conflicts and a dependency function. For the purposes of the formalisation, we have separated these three parts, as not every definition needs all three parts of the tuple.

Since repositories are always finite, we have used Coq's `FSet` library, which is a library for the representation of finite sets. There are other possibilities for sets, such as a representation of linked lists (`ListSet`), and a representation using characteristic functions (`Ensemble`).

The advantages of the `FSet` library over these other implementations is that it has a larger library of basic theorems, which saves quite a lot of time. Furthermore, the `Ensemble` library can also deal with infinite sets; this only complicates matters for our purposes, since repositories are always finite.

Let us start with defining a package type:

```
Module Type PACKAGE.
  Parameter t: Set.

  Parameter eq: t → t → Prop.
  Parameter lt: t → t → Prop.

  Axiom eq_refl: ∀ x: t, eq x x.
  Axiom eq_sym: ∀ x y: t, eq x y → eq y x.
  Axiom eq_trans: ∀ x y z: t, eq x y → eq y z → eq x z.

  Axiom lt_trans: ∀ x y z: t, lt x y → lt y z → lt x z.
  Axiom lt_not_eq: ∀ x y: t, lt x y → ¬ eq x y.

  Parameter compare: ∀ x y: t, Compare lt eq x y.
  Parameter eq_dec: ∀ x y, { eq x y } + { ¬ eq x y }.
End PACKAGE.

Declare Module PACKAGE: PACKAGE.
Declare Module PACKAGESET : FSETINTERFACE.S with Module E := PACKAGE.
Export PackageSet.
```

Note that in the definition of the PACKAGE type, equivalence and comparison relations with their attendant axioms must be specified. What exactly is a package is not of interest here; just the fact that there exists an equality relation on it, as well as a comparison.

After the definition of PACKAGE, the definition of a conflict follows:

```
Module CONFLICT := PAIRORDEREDTYPE PACKAGE PACKAGE.

Declare Module CONFLICTSET : FSETINTERFACE.S with Module E := CONFLICT.

Axiom conflicts_sym: ∀ (C: ConflictSet.t) (p q: Package.t), ConflictSet.In (p, q) C → ConflictSet.In (q, p) C.

Axiom conflicts_irrefl: ∀ (C: ConflictSet.t) (p: Package.t), ¬ ConflictSet.In (p, p) C.
```

In other words, a conflict is a pair of packages (implemented in `FSet` by the `PairOrderedType` module), with the two axioms that correspond to axiom 2.3 from chapter 2.

6.2   Dependencies

With packages and conflicts defined, the next step is a module which deals with the basic definitions. This module will contain all necessary definitions and lemmas needed for the definition of the package cone (definition 2.15). The reason we have put this in a separate module is to parametrise the method of obtaining the cone, so that theorems about different types of cones can be specified: the 'usual' cone, a cone of conjunctive dependencies only, and even the reverse cone (the set of packages on which a given package depends). How this is done in practice will be shown in section 6.5.

First we shall define a variable R, the repository, and D (here called `Dependencies`) for the dependency function:

```
Section dep_cone_stuff.

Variable Dependencies: Package.t → list PackageSet.t.
Axiom Dep_compat_eq: ∀ p q: Package.t, Package.eq p q → Dependencies p = Dependencies q.
Axiom no_self_dep: ∀ p: Package.t, ¬ ∃ d, In p d ∧ List.In d (Dependencies p).
Variable dep_filter: PackageSet.t → bool.
Axiom dep_filter_eq: ∀ p q: PackageSet.t, PackageSet.eq p q → dep_filter p = dep_filter q.

Add Morphism dep_filter: dep_filter_m.
```

Apart from the dependency function, we are adding a second function: `dep_filter`. This will be used later on in order to prove propositions about conjunctive dependencies; the filter is used to exclude specific dependencies, such as dependencies that have more than one alternative. How exactly this works will become clear later on.

The axioms specify that equal packages have equal dependencies, that the dependency filter function has the same result for equal packages, and that a package cannot depend directly on itself (which conforms to axiom 2.2).

The `dep_filter` function is specified as a a *morphism*: this does not add any new information, but enables it to be used more easily. More specifically, Coq now can automatically rewrite something like $\text{dep\_filter}(a)$ into $\text{dep\_filter}(b)$, if it has already been established that $a = b$.

Here follows the definition of `dependency_function`. This function maps a package to all of its direct dependencies within the repository R (note that the alternatives in the co-domain of the `Dependencies` function are not specified as being within the repository; this is intentional, as it is very well possible for a package to have dependencies on packages outside the repository).

```
Definition dependency_function (p: Package.t): PackageSet.t :=
  (List.fold_left (fun alt acc ⇒
    union alt acc
  ) (List.filter dep_filter (Dependencies p)) empty).
```

And then the definition of a direct dependency (see also definition 2.10).

```
Definition direct_dependency (p: Package.t) (q: Package.t) :=
  ∃ d: PackageSet.t, In q d ∧
  List.In d (List.filter dep_filter (List.map (inter R) (Dependencies p))).

Add Morphism direct_dependency with signature Package.eq ⇒ Package.eq ==> iff
  as direct_dependency_m.
```

Here we can see the usage of the dependency filter: q is a dependency of p if and only if there is a d in the dependencies of p that contains q, and if the filter function returns `true` for $d \cup R$.

Additionally, we have added another morphism, for `direct_dependency`. Since Coq's standard equality is not used here, it is necessary to specify a signature: stated here is that for any p, p', q and q', if $p = p'$ and $q = q'$ (according to the equality defined in the `Package` module above), then if $p \twoheadrightarrow q$, then $p' \rightarrow q'$.

Now, let us introduce lemmas for the important properties of these functions. This way, it is easier to use them in proofs without having to unroll their definitions every time.

> Lemma direct_dependency_depfunc: $\forall$ (p: *Package.t*) (q: *Package.t*),
>   direct_dependency $p\ q$ → *In* q (dependency_function p).
>
> Lemma depfunc_direct_dependency: $\forall$ (p: *Package.t*) (q: *Package.t*),
>   *In* q (dependency_function p) → direct_dependency $p\ q$.

With this, it is possible to formalise the notion of a "dependency path"; a list of packages connected by dependencies.

> Function *dependency_path* ($p\ q$: *Package.t*) (l: **list** *Package.t*)
>   { struct l }: Prop :=
>   match l with
> | nil ⇒ direct_dependency $p\ q$
> | h::t ⇒ direct_dependency $p\ h$ ∧ dependency_path $h\ q\ t$
>   end.

In fact, the definition above is not the only way to formalise the notion of a dependency path: it can also be defined 'in reverse':

> Function *rev_dependency_path* ($p\ q$: *Package.t*)
>   (l: **list** (*Package.t*)) { struct l }: Prop :=
>   match l with
> | nil ⇒ direct_dependency $p\ q$
> | h::t ⇒ rev_dependency_path $p\ h\ t$ ∧ direct_dependency $h\ q$
>   end.

Having the two notions can be useful; sometimes a proof is easier to complete using the 'normal' notion, and sometimes the reverse notion is simpler in use. It remains, obviously, necessary to prove that they are equivalent.

For this, we shall first introduce the definition of 'dependency': if there is a dependency path between two packages p and q, according to definition 2.11, p depends on q. Formalised, this becomes:

> Definition dependency ($p\ q$: *Package.t*): Prop :=
> $\exists\ l$: **list** (*Package.t*), dependency_path $p\ q\ l$.
>
> Definition rev_dependency ($p\ q$: *Package.t*): Prop :=
> $\exists\ l$: **list** (*Package.t*), rev_dependency_path $p\ q\ l$.

The notion of equivalence then becomes:

> Lemma dep_rev_dep: $\forall\ p\ q$,
>   rev_dependency $p\ q$ ↔ dependency $p\ q$.

In order to prove this equivalence, two lemmas about combining two dependency paths into one are needed:

> Lemma dp_split: $\forall\ p\ q\ r\ l\ l'$,
>   dependency_path $p\ q\ l$ → dependency_path $q\ r\ l'$ → dependency_path $p\ r$ (l++(q::l')).
>
> Lemma rev_dp_split: $\forall\ p\ q\ r\ l\ l'$,
>   rev_dependency_path $p\ q\ l$ → rev_dependency_path $q\ r\ l'$ → rev_dependency_path $p\ r$
> (l'++(q::l)).

Using this, it becomes easy to prove that the dependency relation introduced above is transitive:

```
Lemma dependency_trans:
  ∀ (p q r: Package.t), dependency p q → dependency q r → dependency p r.
```

For the dependency cone, later on, a definition of a dependency path that consists only of packages from a specific repository will be needed:

```
Function dependency_path_in (R: PackageSet.t) (p q: Package.t)
  (l: list (Package.t)) { struct l }: Prop :=
  match l with
  | nil ⇒ In p R ∧ In q R ∧ direct_dependency p q
  | h::t ⇒ In p R ∧ direct_dependency p h ∧ dependency_path_in R h q t
  end.

Definition dependency_in (R: PackageSet.t) (p q: Package.t): Prop :=
  ∃ l: list (Package.t), dependency_path_in R p q l.
```

with some attendant lemmas:

```
Lemma dp_R: ∀ R p q l,
  dependency_path_in R p q l → In q R.

Lemma dp_in_dp: ∀ R p q l, dependency_path_in R p q l → dependency_path p q l.
```

Another function that is useful is the equivalent of `dependency_function` for a set, i.e. given a set P, the union of P and all `dependency_function`(p) for $p \in P$. (this is the function that, if iterated repeatedly until obtention of a fixpoint, results in the dependency cone of a set of packages). The function is easy to define using FSet's `fold` function:

```
Definition dependencies (P: PackageSet.t): PackageSet.t :=
  fold (fun p acc ⇒
    union (dependency_function p) acc
  ) P P.
```

Like the earlier dependency function, its properties follow immediately after its definition; in this case, the fact that for any element $q \in$ dependencies(P), there is a package $p \in P$ such that $p \twoheadrightarrow q$ and vice versa.

```
Lemma dependencies_dependency:
  ∀ (q: Package.t) (P: PackageSet.t),
  In q (dependencies P) → In q P ∨ Exists (fun p ⇒ direct_dependency p q) P.

Lemma dependency_dependencies:
  ∀ (q: Package.t) (P: PackageSet.t),
  Exists (fun p ⇒ direct_dependency p q) P → In q (dependencies P).
```

This function is monotone with respect to the subset relation:

```
Lemma dependencies_monotone: ∀ (P: PackageSet.t) ,
  P [≤] dependencies P.
```

And furthermore, it preserves the subset relation.

```
Lemma dependencies_subset: ∀ (P Q: PackageSet.t), P [≤] Q → dependencies P [≤] dependencies Q.
```

6.3   The dependency cone

The definition of the dependency cone merits its own section, mostly because it is the part that presented the most difficulties.

As seen in chapter 2, the definition in itself is easy enough (see definition 2.15); the transitive closure of the dependency relationship.

In terms of the Coq definitions seen so far, the cone can be defined as a repetitive application of the `dependencies` function until obtention of a fixpoint. This is guaranteed to terminate, since the `dependencies` function always is a subset of the repository, which is finite.

In a Coq recursive function specification, the `measure` keyword can be used to indicate a natural number that strictly decreases with every iteration of the function. In this case, that is the difference between the number of packages in the repository and the number of packages in the cone.

It is for this reason that it is not possible to just take the cone of any set, but only of a subset of a repository: in order to guarantee termination, an upper limit for the size of the cone is needed, which is the size of the repository.

```
Function cone (P: {x : PackageSet.t | x [≤] R})
  { measure (fun x ⇒ cardinal R - cardinal (proj1_sig P)) P }: PackageSet.t :=
  if equal (inter R (dependencies (proj1_sig P))) (proj1_sig P)
  then (proj1_sig P)
  else
    cone (exist (fun v ⇒ v [≤] R) (inter R (dependencies (proj1_sig P))))
    (fun a ⇒ inter_subset_1 (s:=R) (s':=dependencies (proj1_sig P)) (a:=a))).
```

This definition looks rather complicated, but upon closer inspection, it's actually fairly straightforward, once one gets past the syntax.

First the argument specification. The function `cone` takes an argument P, which has as its type something that is a `PackageSet`, with an extra specification that this package set is a subset of R. Any element of this type thus has to contain both a package set, and a proof that this package set is a subset of R, as we will see later on.

The function itself is a fixpoint declaration; one specifies a function and proves that after a finite number of applications, it provides a result. The `measure` keyword discussed before can be used for this.

The measure used here is $|R| - |\text{dependencies}(P)|$. This, by the way, is the reason why the type of P is not simply `Set`; if there is no upper limit to the size of P, there is no way to give a descending measure. Coq automatically generates the resulting proof obligations.

Now the function itself can be defined (the function that is going to be applied repeatedly until a fixpoint is reached; Coq automatically renames this to `cone_F`). Two cases are distinguished: if $R \cup \text{dependencies}(P) = P$, P is returned (the fixpoint is reached); otherwise, the cone of $R \cup \text{dependencies}(P)$ is returned.

The reason for specifying this using `exist` is again the specification type: the parameter of `cone` is not a simple package set, but a package set that is a subset of R. The `exist` function creates this specification type: first a pattern (in this case "v is a subset of R"), then the package set, and then a proof term for the fact that the package set complies with the pattern (using the `inter_subset_1`

lemma, which specifies that $\forall_{ss'a}[a \in s \cap s' \rightarrow a \in s]$). Owing to the syntactic specificities of Coq, it is necessary to specify the parameters s, s' and a explicitly.

The resulting function cone_F is rather complicated, so in order to simplify consequent proofs, let us immediately define its properties: if a package q is an element of the dependency cone of p, there must be a dependency path from p to q and vice versa.

Lemma dep_cone: $\forall$ (P: *PackageSet.t* | P [$\leq$] R) *q,*
   Exists (fun $p \Rightarrow$ dependency_in R p q) (proj1_sig P) $\rightarrow$ *In q* (cone P).

Lemma cone_dep: $\forall$ (P: *PackageSet.t* | P [$\leq$] R) *q,*
   In q (cone P) $\rightarrow$
   In q (proj1_sig P) $\vee$ Exists (fun $p \Rightarrow$ dependency_in R p q) (proj1_sig P).

For these two proofs, a few lemmas are needed. These lemmas use the iter and cone_F functions; the cone_F function is the body of the cone function (the if statement from its definition), and the iter function is used to apply this function a certain number of times.

The definitions are fairly straightforward, even though there are some extra arguments that should not hinder comprehension (they are necessary for dealing with the specification type of P):

Lemma iter_cone_monotone: $\forall$ P n,
   iter (\{x : t | x [$\leq$] R\} $\rightarrow$ t) n cone_F (fun $v \Rightarrow$ proj1_sig v) P [$\leq$]
   iter (\{x : t | x [$\leq$] R\} $\rightarrow$ t) (S n) cone_F (fun $v \Rightarrow$ proj1_sig v) P.

Lemma iter_cone_expanding:
   $\forall$ P k n, $k \leq n \rightarrow$
   iter (\{x | x [$\leq$] R\} $\rightarrow$ t) k cone_F (fun $v \Rightarrow$ proj1_sig v) P [$\leq$]
   iter (\{x | x [$\leq$] R\} $\rightarrow$ t) n cone_F (fun $v \Rightarrow$ proj1_sig v) P.

Lemma dep_path_iter: $\forall$ P p q n,
   In p (proj1_sig P) $\rightarrow$ dependency_path_in R p q n $\rightarrow$
   In q (iter (\{x | x [$\leq$] R\} $\rightarrow$ t) (S (length n)) cone_F (fun $v \Rightarrow$ proj1_sig v) P).

Now, after the definitions of the characteristic functions of cone, some some simple properties can be established; the cone is always a subset of the repository, the cone of a set P of packages always is a superset of P, and if $P \subseteq Q$, then $\text{cone}(P) \subseteq \text{cone}(Q)$.

Lemma cone_subset: $\forall$ (P: *PackageSet.t* | P [$\leq$] R),
   (proj1_sig P) [$\leq$] cone P.

Lemma cone_subset_R: $\forall$ (P: *PackageSet.t* | P [$\leq$] R),
   cone P [$\leq$] R.

Lemma cone_of_subset_is_subset: $\forall$ (P1: *PackageSet.t* | P1 [$\leq$] R)
(P2: *PackageSet.t* | P2 [$\leq$] R),
(proj1_sig P1) [$\leq$] (proj1_sig P2) $\rightarrow$ cone P1 [$\leq$] cone P2.

This concludes the module PkgCone.

6.4 Repository properties

Now that the preliminaries are out of the way, we can start formalising some proofs from chapters 2 and 3.

To start with, let us introduce the notion of a conjunctive dependency:

```
Definition is_conjunctive (a: PackageSet.t) :=
    ∃ p: Package.t, a [=] (singleton p).

Lemma conjunctive_dec: ∀ a: PackageSet.t,
    { is_conjunctive a } + { ¬ is_conjunctive a }.

Definition is_conjunctive_bool (a: PackageSet.t): bool :=
    if conjunctive_dec a then true else false.
```

Then, the notions of abundance and peace are specified, as per definitions 2.4 and 2.6 from chapter 2. First, a package p is satisfied (with respect to a set S) if all its dependencies are satisfied in S:

```
Definition satisfied_pkg (S: PackageSet.t) (p: Package.t): Prop :=
    ∀ d: PackageSet.t, List.In d (Dependencies p) →
    ∃ p': Package.t, In p' (inter S d).

Definition satisfied_pkg_bool (S: PackageSet.t) (p: Package.t): bool :=
    forallb (fun d ⇒ exists_ (fun p' ⇒ true) (inter S d))
        (Dependencies p).

Lemma spb_ok: ∀ (S: PackageSet.t) (p: Package.t),
    satisfied_pkg S p ↔ Is_true (satisfied_pkg_bool S p).
```

Note that there is also boolean version of the definition; this can come in useful in order to use this formalisation when proving properties of actual programs; this is discussed in more detail in the 'Future work' section of the conclusion. There is also a proof that shows that both versions are equivalent.

The fact that the `satisfied_pkg` predicate is decidable follows easily from the fact that its boolean version must necessarily be `true` or `false`:

```
Lemma satisfied_dec: ∀ (S: PackageSet.t) (p: Package.t),
    decidable (satisfied_pkg S p).
```

A few useful lemmas about satisfaction:

```
Lemma satisfied_union1:
    ∀ (S S': PackageSet.t) (p: Package.t),
        satisfied_pkg S p → satisfied_pkg (union S S') p.

Lemma satisfied_union2:
    ∀ (S S': PackageSet.t) (p: Package.t),
        satisfied_pkg S' p → satisfied_pkg (union S S') p.

Lemma satisfied_subset:
    ∀ (S S': PackageSet.t) (p: Package.t),
        S [≤] S' → satisfied_pkg S p → satisfied_pkg S' p.
```

Now, a set is abundant if all of its elements are satisfied. Additionally, there is a proof of the fact that abundance is a morphism (abundance is preserved under set equality), and that it is decidable.

```
Definition abundant (S: PackageSet.t): Prop :=
   PackageSet.For_all (satisfied_pkg S) S.
```

Add *Morphism* abundant with *signature* eq ⇒ iff as *abundant_m*.

```
Lemma abundant_dec: ∀ S: PackageSet.t,
   decidable (abundant S).
```

The formalisation of corollary 2.5:

```
Lemma abundant_union:
   ∀ (S S': PackageSet.t),
      abundant S → abundant S' → abundant (PackageSet.union S S').
```

For the formalisation of peace, the concept of being *concerned* is introduced; a conflict (c_1, c_2) is concerned with a set S if and only if both c_1 and c_2 are in S. Concernedness, too, is preserved under equality (both package and set equality), and it has a boolean version as well.

```
Definition concerned (S: PackageSet.t) (c: Package.t × Package.t): Prop :=
   match c with
   | (p, q) ⇒ (In p S) ∧ (In q S)
   end.
```

Add *Morphism* concerned with *signature* PackageSet.eq ⇒ Conflict.eq ==> iff as *concerned_m*.

```
Definition concerned_bool (S: PackageSet.t) (c: Package.t × Package.t): bool :=
   match c with
   | (p, q) ⇒ PackageSet.mem p S && PackageSet.mem q S
   end.
```

```
Lemma concerned_dec:
   ∀ (S: PackageSet.t) (c: Package.t × Package.t),
   decidable (concerned S c).
```

```
Lemma concerned_ok: ∀ (S: PackageSet.t) (c: Package.t × Package.t),
   concerned S c ↔ Is_true (concerned_bool S c).
```

Now it becomes easy to define peace as the absence of concerned conflicts:

```
Definition peaceful (S: PackageSet.t) (C: ConflictSet.t): Prop :=
   ConflictSet.For_all (fun c ⇒ ¬ (concerned S c)) C.
```

Add *Morphism* peaceful with *signature* eq ⇒ ConflictSet.eq ==> iff as *peaceful_m*.

```
Lemma peaceful_dec:
   ∀ (S: PackageSet.t) (C: ConflictSet.t),
      decidable (peaceful S C).
```

Any subset of a peaceful set is also peaceful:

```
Lemma peaceful_subset: ∀ (S1 S2: PackageSet.t) (C: ConflictSet.t),
   S1 [≤] S2 → peaceful S2 C → peaceful S1 C.
```

As we can see, things start to converge towards the formalisation and proof of theorem 3.25. One more lemma: if a set is not peaceful, there is a specific conflict to be 'blamed' for that:

> Lemma blame_conflict: ∀ (*I*: *PackageSet.t*) (*C*: *ConflictSet.t*),
>   ¬ peaceful *I C* →
>   ConflictSet.Exists (fun *c* ⇒ concerned *I c*) *C*.

With this, lemma 3.23 can be proved, which is one of the substantive ingredients for the proof of theorem 3.25.

> Lemma not_peaceful_conflict:
>   ∀ (*S S′*: *PackageSet.t*) (*C*: *ConflictSet.t*),
>   (peaceful *S C*) → (peaceful *S′ C*) → ¬ (peaceful (*union S S′*) *C*) →
>   Exists (fun *p* ⇒ Exists (fun *q* ⇒ *ConflictSet.In* (*p*, *q*) *C*) *S′*) *S*.

After this, healthiness can be defined as a combination of abundance and peace. Obviously, healthiness is preserved under equality and decidable.

> Definition healthy (*S*: *PackageSet.t*) (*C*: *ConflictSet.t*): Prop :=
>   abundant *S* ∧ peaceful *S C*.
>
> Add *Morphism* healthy with *signature* eq ⇒ ConflictSet.eq ==> iff as *healthy_m*.
>
> Lemma healthy_dec: ∀ (*S*: *PackageSet.t*) (*C*: *ConflictSet.t*),
>   decidable (healthy *S C*).

An empty set is healthy.

> Lemma empty_healthy: ∀ (*S*: *PackageSet.t*) (*C*: *ConflictSet.t*),
>   Empty *S* → healthy *S C*.

6.5   Installability

This section is about the definition of installability and co-installability; see also definitions 2.8 and 2.9 from chapter 2.

> Definition installable (*R*: *PackageSet.t*) (*C*: *ConflictSet.t*) (*p*: *Package.t*) :=
>   ∃ *I*: *PackageSet.t*, *I* [≤] *R* ∧ *In p I* ∧ healthy *I C*.
>
> Definition is_install_set (*p*: *Package.t*) (*R*: *PackageSet.t*) (*C*: *ConflictSet.t*) (*I*: *Package-Set.t*) :=
>   *In p I* ∧ *I* [≤] *R* ∧ healthy *I C*.
>
> Definition co_installable (*R*: *PackageSet.t*) (*C*: *ConflictSet.t*) (*S*: *PackageSet.t*) :=
>   ∃ *I*: *PackageSet.t*, *I* [≤] *R* ∧ *S* [≤] *I* ∧ healthy *I C*.

Fairly trivial: if the package *p* is installable, then the set {*p*} is co-installable.

> Lemma inst_coinst: ∀ (*R*: *PackageSet.t*) (*C*: *ConflictSet.t*) (*p*: *Package.t*),
>   installable *R C p* ↔ co_installable *R C* (*singleton p*).

Let us start by defining the difference between normal dependencies (i.e. all specified dependencies) and conjunctive dependencies (single dependencies). This is done by using the dependency filter mentioned previously: for normal dependencies, the filter that always returns true (thus selecting all dependencies) is used, and for conjunctive dependencies, the is_conjunctive_bool function defined before is used.

```
Definition direct_normal_dependency (p: Package.t) (q: Package.t) :=
    direct_dependency Dependencies (fun x ⇒ true) p q.

Definition direct_conjunctive_dependency (p: Package.t) (q: Package.t) :=
    direct_dependency Dependencies is_conjunctive_bool p q.
```

If there is a conjunctive direct dependency, there is a normal direct dependency as well:

```
Lemma conj_dep_is_dep:
    ∀ p q, direct_conjunctive_dependency p q → direct_normal_dependency p q.
```

The same applies for dependency paths:

```
Definition normal_dependency_path (p q: Package.t)
    (l: list (Package.t)): Prop :=
    dependency_path Dependencies (fun a ⇒ true) p q l.

Definition conjunctive_dependency_path (p q: Package.t)
    (l: list (Package.t)): Prop :=
    dependency_path Dependencies is_conjunctive_bool p q l.

Lemma conj_dp_is_dp: ∀ p q l,
    conjunctive_dependency_path p q l → normal_dependency_path p q l.
```

And finally, the definitions of the normal and conjunctive dependency relationship, as well as the 'normal' dependency cone.

```
Definition normal_dependency (p q: Package.t): Prop :=
    dependency Dependencies (fun a ⇒ true) p q.

Definition conjunctive_dependency (R: PackageSet.t) (p q: Package.t): Prop :=
    dependency Dependencies is_conjunctive_bool p q.

Definition normal_cone (R: PackageSet.t) (S: PackageSet.t | S [≤] R):=
    cone Dependencies (fun a ⇒ true) R S.
```

All this can be combined into the following theorem: if a package p is installable with respect to a repository R, it is also installable with respect to $\Delta_R(p)$: this is proposition 2.17.

```
Lemma installable_in_cone:
    ∀ (R: PackageSet.t) (C: ConflictSet.t) (P: PackageSet.t | P [≤] R),
    co_installable R C (proj1_sig P) →
    co_installable (normal_cone R (exist (fun v ⇒ v [≤] R) (proj1_sig P) (proj2_sig P))) C
(proj1_sig P).
```

Next in line is proposition 2.18; any conjunctive dependency of p is always part of the install set of p:

```
Lemma conjunctive_always_installed:
    ∀ R C p q I,
        conjunctive_dependency R p q →
        is_install_set p R C I →
        In q I.
```

In the same vein, a package that conjunctively depends on a non-installable

package is not-installable itself:

> Lemma not_installable_conjunctive: ∀ (R: *PackageSet.t*) (C: *ConflictSet.t*)
> (*p q*: *Package.t*),
> ¬ installable *R C q* → conjunctive_dependency *R p q* → ¬ installable *R C p*.

6.6 Strong dependencies and conflicts

Now, strong dependencies are formalised as per definition 3.1.

> Definition strong_dep (R: *PackageSet.t*) (C: *ConflictSet.t*) (p: *Package.t*) (q: *Package.t*)
> :=
> (∃ *N*: *PackageSet.t*, is_install_set *p R C N*) ∧
> ∀ *I*: *PackageSet.t*, *I* [≤] *R* → healthy *I C* ∧ *In p I* → *In q I*.

The strong dependency relationship is transitive:

> Lemma strong_dep_trans:
> ∀ (R: *PackageSet.t*) (C: *ConflictSet.t*) (*p q r*: *Package.t*),
> strong_dep *R C p q* ∧ strong_dep *R C q r* → strong_dep *R C p r*.

Historically, strong dependencies were defined differently: *p* depends strongly on *q* if and only if *p* is installable, and there is no install set of *p* that does not include *q*. This negative definition is equivalent to the positive one from definition 3.1:

We have opted to use the positive version of the definition as the standard definition, because it is constructive and therefore allows for more straightforward proofs. The negative version would need many proofs by absurdity (indeed, the proof that the negative version implies the positive version is an example of this), which are not as informative.

> Definition strong_dep_neg (R: *PackageSet.t*) (C: *ConflictSet.t*) (p: *Package.t*) (q: *Package.t*) :=
> (∃ *N*: *PackageSet.t*, is_install_set *p R C N*) ∧
> ¬ ∃ *I*: (*PackageSet.t*), *I* [≤] *R* ∧ healthy *I C* ∧ *In p I* ∧ ¬ *In q I*.
>
> Lemma strong_dep_pos_neg:
> ∀ (R: *PackageSet.t*) (C: *ConflictSet.t*) (*p q*: *Package.t*),
> strong_dep *R C p q* → strong_dep_neg *R C p q*.
>
> Lemma strong_dep_neg_pos:
> ∀ (R: *PackageSet.t*) (C: *ConflictSet.t*) (*p q*: *Package.t*),
> strong_dep_neg *R C p q* → strong_dep *R C p q*.

Follows the definition of strong conflicts, as per definition 3.21.

> Definition strong_conflict (R: *PackageSet.t*) (C: *ConflictSet.t*) (p: *Package.t*) (q: *Package.t*) :=
> installable *R C p* ∧ installable *R C q* ∧
> ¬ ∃ *I*: *PackageSet.t*, healthy *I C* ∧ *In p I* ∧ *In q I*.

With these lemmas, theorem 3.25 can be proved. This proof is somewhat larger, so we shall show it in full:

```
Theorem scp: ∀ (R: PackageSet.t) (C: ConflictSet.t) (p: Package.t | In p R) (q: Package.t |
In q R),
    strong_conflict R C (proj1_sig p) (proj1_sig q) →
    ConflictSet.Exists (fun c ⇒
        match c with
            (c1,c2) ⇒ (E.eq (proj1_sig p) c1 ∨ normal_dependency (proj1_sig p) c1) ∧
                        (E.eq (proj1_sig q) c2 ∨ normal_dependency (proj1_sig q) c2)
        end) C.
```

```
intros R C pp qq H. destruct pp as [p Hp]. destruct qq as [q Hq]. unfold strong_conflict
in H.
    destruct H as [Hpi [Hqi HI]].
```

Here, proposition 2.17 is used twice; there are P and Q as installation sets of p and q, so $\Delta(p) \cap P$ and $\Delta(q) \cap Q$ are installation sets too.

The original P and Q are no longer needed, so in the following, for ease of notation, $\Delta(p) \cap P$ will be referred to as P, and $\Delta(q) \cap Q$ as Q.

```
    destruct (installable_in_cone R C (exist (fun v ⇒ v [≤] R) (singleton p) (s_ss Hp))) as [P
[HP [HpP HPh]]].
        apply → inst_coinst. apply Hpi. simpl in HP. simpl in HpP.
    destruct (installable_in_cone R C (exist (fun v ⇒ v [≤] R) (singleton q) (s_ss Hq))) as [Q
[HQ [HqQ HQh]]].
        apply → inst_coinst. apply Hqi. simpl in HQ. simpl in HqQ.
```

Then lemma 3.23 is applied. The union of P and Q is not peaceful, so there must be a conflict (p', q') with $p \in P$ and $q \in Q$.

```
    destruct (not_peaceful_conflict P Q C) as [p' [Hp' [q' [Hq' HC]]]].
        apply HPh.
        apply HQh.
        intro. apply HI. exists (union P Q). split. split.
            apply abundant_union. apply HPh. apply HQh. apply H. split.
            apply union_2. apply HpP. apply singleton_2. reflexivity.
            apply union_3. apply HqQ. apply singleton_2. reflexivity.
```

Now, (p', q') is a conflict. Furthermore, p' is in the cone of p, so $p \twoheadrightarrow p'$, and in the same vein, $q \twoheadrightarrow q'$. All this means that the conflict (p', q') is the needed conflict. The lemma `cone_dep` from the previous section is used to prove that p' is a dependency of p (and q' of q).

```
    exists (p′, q′). split. apply HC. split.
      elim (cone_dep Dependencies (fun a ⇒ true) R (exist (fun v ⇒ v [≤] R) (singleton p)
(s_ss Hp)) p′).
        intros. left. apply singleton_1. apply H.
        intros. right. destruct H as [x [Hx Hxdep]].
      apply ← (dependency_compat_eq Dependencies (fun _ ⇒ true) p x p′).
      apply dependency_in_dependency with R. apply Hxdep.
      apply singleton_1. apply Hx. apply HP. apply Hp′.
      elim (cone_dep Dependencies (fun a ⇒ true) R (exist (fun v ⇒ v [≤] R) (singleton q)
(s_ss Hq)) q′).
        intros. left. apply singleton_1. apply H.
        intros. right. destruct H as [x [Hx Hxdep]].
      apply ← (dependency_compat_eq Dependencies (fun _ ⇒ true) q x q′).
      apply dependency_in_dependency with R. apply Hxdep.
      apply singleton_1. apply Hx. apply HQ. apply Hq′.
```

6.7 Triangle conflicts

The previous theorem being proven, let us prepare ourselves for theorem 3.28. The proof starts with introducing the notion of reverse dependencies.

```
Variable ReverseDependencies: Package.t → list PackageSet.t.
Axiom RevDep: ∀ (f: t → bool) (p q: Package.t),
  In p (dependency_function Dependencies f q) ↔
  In q (dependency_function ReverseDependencies f p).
Definition reverse_dependency_function (f: t → bool) (p: Package.t): PackageSet.t :=
  dependency_function ReverseDependencies f p.
```

If p depends on q, then q reverse-depends on p:

```
Lemma pred_ok: ∀ (f: t → bool) (p q: Package.t),
  direct_dependency Dependencies f p q ↔
  In p (reverse_dependency_function f q).
```

Let us introduce a new dependency function, in order to remove superfluous dependencies as per definition 3.27 in chapter 3.

```
Definition NoSupDependencies (R: PackageSet.t) (p: Package.t): list (PackageSet.t) :=
  List.filter (fun dep ⇒ negb (existsb (fun a ⇒ strict_subset a dep) (Dependencies p))) (Dependencies p).
```

The first thing to prove is that this does not hinder installability. For this, a function is introduced that retains only the smallest subdependency of its argument in a list of dependencies, and proves that the result of this function is indeed a list that does not contain superfluous dependencies:

```
Definition find_smallest_subdep (l: list PackageSet.t) (d: PackageSet.t): PackageSet.t :=
  List.fold_left (fun acc d' ⇒
    if strict_subset d' acc then d' else acc
  ) l d.
```

```
Lemma smallest_subdep_is_subset:
  ∀ l d, find_smallest_subdep l d [≤] d.
```

```
Lemma smallest_subdep_in_list:
  ∀ l d, find_smallest_subdep l d = d ∨ List.In (find_smallest_subdep l d) l.
```

```
Lemma ss_ok: ∀ (l: list PackageSet.t) (d: PackageSet.t),
  ¬ ∃ d': PackageSet.t, List.In d' l ∧ d' [<] (find_smallest_subdep l d).
```

Follows the proof that removing superfluous dependencies does not influence installability.

```
Lemma NSD_installability:
  ∀ (R: PackageSet.t) (C: ConflictSet.t) (p: Package.t), In p R →
  (installable Dependencies R C p ↔ installable (NoSupDependencies R) R C p).
```

Then, the definition of a triangle conflict.

```
Definition is_triangle (R: PackageSet.t) (f: t → bool) (c: Conflict.t): Prop :=
  let (c1, c2) := c in
  ∃ t,
    (∃ d, List.In d (Dependencies t) ∧ In c1 d ∧ In c2 d ∧
      (∀ d', (∃ t', List.In d' (Dependencies t')) ∧ (In c1 d' ∨ In c2 d') →
        d [=] d')).
```

To start with the proof of theorem 3.28, here is the condition: all concerned conflicts are triangle conflicts.

```
Definition only_triangles (R: PackageSet.t) (C: ConflictSet.t) (f: t → bool) (A: PackageSet.t
| A [≤] R) :=
  ConflictSet.For_all (fun c ⇒ concerned
    (cone (NoSupDependencies R) f R A) c →
      is_triangle R f c ) C.
```

Also needed is a function K as specified in the proof of theorem 3.28. Here, it is called packages_to_remove:

```
Definition packages_to_remove (R: PackageSet.t) (C: ConflictSet.t) (A: PackageSet.t) (B:
PackageSet.t) :=
  PackageSet.filter (fun p ⇒
    ConflictSet.exists_ (fun c ⇒ let (c1, c2) := c in
      PackageSet.mem c1 A && PackageSet.mem c2 B && beq_pkg p c1
    ) C
  ) (union A B).
```

```
Lemma triangles_ok:
   ∀ (R: PackageSet.t) (C: ConflictSet.t) (a: PackageSet.t | a [≤] R),

   (For_all (fun a ⇒ installable (NoSupDependencies R) R C a) (proj1_sig a)) →
   only_triangles R C (fun _ ⇒ true) a →
   ¬ ConflictSet.Exists (fun c ⇒ let (c1, c2) := c in In c1 (proj1_sig a) ∨ In c2 (proj1_sig
a)) C →
   co_installable (NoSupDependencies R) R C (proj1_sig a).
```

We shall give this proof in full as well:

```
intros R C aa. destruct aa as [a Ha]. simpl. intros Hsep Htri Hcc.
   unfold co_installable. unfold For_all in Hsep.
   induction a using set_induction.
```

The set a is a set of packages, such that $\Delta(a)$ does not contain any conflicts. The proof proceeds by induction on a. The first case, consequently, is the case $a = \emptyset$. This is fairly easy; the empty set is a peaceful and abundant set that contains all elements of $\emptyset$.

```
   exists empty. split. apply subset_empty. split. rewrite (empty_is_empty_1 H).
   reflexivity. split. unfold abundant. unfold For_all. intros. absurd (In x empty).
   intro. apply → empty_iff. apply H1. apply H0.
   unfold peaceful. unfold ConflictSet.For_all. intros. unfold concerned.
   destruct x. intro. destruct H1. absurd (In t0 empty).
   intro. apply → empty_iff. apply H3. apply H1.
```

Then the induction. There is a co-installable set a_1. To be proven: that $a_1 \cup \{x\}$ is co-installable as well. It is known that x itself is installable. This means that there exist sets A_1 and A_x (the install sets of a_1 and x respectively, which per proposition 2.17 are part of the cone of a_1 or x). Note that a_1 is a subset of the repository R, and that $x \in R$.

```
   assert (a1 [≤] R) as Ha1. unfold Subset. intros. apply Ha. elim (H0 a). intros.
apply H3. right. apply H1.
   elim (installable_in_cone (NoSupDependencies R) R C (exist (fun v ⇒ v [≤] R) a1 Ha1)).
   simpl. intros A1 X. destruct X as [HA1_1 [HA1_2 [HA1_A HA1_P]]].
   assert (A1 [≤] R) as HA1. unfold Subset. intros.
   apply (cone_subset_R (NoSupDependencies R) (fun _ ⇒ true) R (exist (fun v ⇒ v [≤]
R) a1 Ha1)).
   apply HA1_1. apply H1.
   assert (In x R) as Hx. unfold Subset. apply Ha. elim (H0 x). intros. apply H2.
left. reflexivity.
   elim (installable_in_cone (NoSupDependencies R) R C (exist (fun v ⇒ v [≤] R)
(singleton x) (s_ss Hx))).
   simpl. intros Ax X. destruct X as [HAx_1 [HAx_2 [HAx_A HAx_P]]].
   assert (Ax [≤] R) as HAx. unfold Subset. intros.
   apply (cone_subset_R (NoSupDependencies R) (fun _ ⇒ true) R (exist (fun v ⇒ v [≤]
R) (singleton x) (s_ss Hx))).
   apply HAx_1. apply H1.
```

Now, installation set for $a_1 \cup \{x\}$ has been found. This set is $A_1 \cup A_x \setminus K(A_1, A_x)$. Therefore, now, it must be proven that this set is a subset of R and that it contains $a_1 \cup \{x\}$:

```
     exists (diff (union A1 Ax) (packages_to_remove R C A1 Ax)). split.
       apply subset_diff. apply (union_subset_3 HA1 HAx).
       split. unfold Subset. intros. apply ← diff_iff. split.
         elim (proj1 (iff_and (H0 a)) H1).
           intros. apply union_3. apply HAx_2. apply singleton_2. apply H2.
           intros. apply union_2. apply HA1_2. apply H2.
         apply Hcc. exists (t0, t1). split.
           apply H3. left. apply In_1 with a. apply → beq_pkg_eq. symmetry. apply H6.
apply H1.
       split.
```

Secondly, that the set is abundant (any dependency d is satisfied):

```
           unfold abundant. unfold For_all. intros. unfold satisfied_pkg. intros.
           rewrite diff_iff in H1. destruct H1.
```

Used here is the fact that A_1 and A_x are abundant, and therefore their union is abundant as well (cf. corollary 2.5). In other words; the aforementioned dependency d is satisfiable in either A_1 or A_x, by a package x_1 ($x_1 \in d \cap (A_1 \cup A_x)$).

```
           elim (abundant_union (NoSupDependencies R) A1 Ax HA1_A HAx_A x0 H1 d).
             intros x1 Hx1.
```

Now, either $x_1 \in K(A_1, A_x)$, or not. The first case is that $x_1 \in K(A_1, A_x)$. This means that there is a conflict (t_0, t_1) with $t_0 \in A_1$, $t_1 \in A_x$ and $t_0 = x_1$.

```
             elim (In_dec x1 (packages_to_remove R C A1 Ax)).
               intros. unfold packages_to_remove in a. rewrite (filter_iff (union A1 Ax)
x1 (compat_bool_3 C A1 Ax)) in a.
               destruct a. rewrite ← (CFacts.exists_iff C (compat_bool_2 A1 Ax x1)) in
H5.
               destruct H5. destruct H5. destruct x2. symmetry in H6. destruct
(andb_true_eq _ _ H6).
               clear H6. destruct (andb_true_eq _ _ H7). clear H7.
```

It has already been established that (t_0, t_1) is a triangle conflict.

```
               destruct (Htri (t0, t1) H5). unfold concerned. split.
               apply (cone_of_subset_is_subset _ _ R (exist (fun v ⇒ v [≤] R) a1 Ha1)).
               simpl. unfold Subset. intros. elim (H0 a). intros. apply H11. right.
apply H7.
               apply HA1_1. apply ← mem_iff. symmetry. apply H6.
               apply (cone_of_subset_is_subset _ _ R (exist (fun v ⇒ v [≤] R) (singleton x)
(s_ss Hx))).
               simpl. unfold Subset. intros. elim (H0 a). intros. apply H11. left. apply
singleton_1. apply H7.
               apply HAx_1. apply ← mem_iff. symmetry. apply H9.
             destruct H7. destruct H7. destruct H10. destruct H11.
             assert (x3 [=] d). apply H12.
               split. exists x0. unfold NoSupDependencies in H2. rewrite filter_In in
H2. destruct H2.
                 apply H2. left. apply In_1 with x1. apply → beq_pkg_eq. symmetry. apply
H8. apply (inter_2 Hx1).
```

Then, it can be proven that $t_1 \in d \cap ((A_1 \cup A_x) \setminus K(A_1, A_x))$; in other words, that t_1 satisfies d.

```
          exists t1. apply inter_3.
            apply ← diff_iff. split.
            apply union_3. apply ← mem_iff. symmetry. apply H9.
            intro. unfold packages_to_remove in H14. rewrite (filter_iff (union A1 Ax)
  t1 (compat_bool_3 C A1 Ax)) in H14.
            destruct H14. rewrite ← (CFacts.exists_iff C (compat_bool_2 A1 Ax t1))
  in H15.
            destruct H15. destruct H15. destruct x4. symmetry in H16. destruct
  (andb_true_eq _ _ H16).
            clear H16. destruct (andb_true_eq _ _ H17).
            apply (HAx_P (t2, t3) H15). split. apply In_1 with t1.
              apply → beq_pkg_eq. symmetry. apply H18.
              apply ← mem_iff. symmetry. apply H9.
              apply ← mem_iff. symmetry. apply H19.
              rewrite ← H13. apply H11.
```

If $x_1 \notin K(A_1, A_x)$, then proving abundance is easy.

```
          intros. exists x1. apply inter_3.
            apply ← diff_iff. split.
            apply (inter_1 Hx1). apply b. apply (inter_2 Hx1). apply H2.
```

Now it must be proven that our set $(A_1 \cup A_x) \setminus K(A_1, A_x)$ is peaceful. This means that the fact must be proven that there is a conflict in $(A_1 \cup A_x) \setminus K(A_1, A_x)$ leads to a contradiction. Therefore, let us assume that there is such a conflict:

```
          unfold peaceful. unfold ConflictSet.For_all. intros. destruct x0. intro.
          unfold concerned in H2. destruct H2. rewrite diff_iff in H2. destruct H2.
          rewrite diff_iff in H3. destruct H3. destruct (union_1 H2). destruct
  (union_1 H3).
```

Having such a conflict means that there is a conflict (t_0, t_1), with $t_0 \in A_1 \cup A_x$, $t_1 \in A_1 \cup A_x$ and neither an element of $K(A_1, A_x)$. The proof proceeds by cases. The case that $t_0 \in A_1$ and $t_1 \in A_1$ can be excluded, because A_1 is peaceful:

```
          apply (HA1_P (t0, t1)). apply H1. split. apply H6. apply H7.
```

Then the case that $t_0 \in A_1$ and $t_1 \in A_x$. This leads to a contradiction, because in that case, t_0 would be an element of $K(A_1, A_x)$:

```
          apply H4. unfold packages_to_remove. rewrite (filter_iff (union A1 Ax) t0
  (compat_bool_3 C A1 Ax)).
          split. apply H2. rewrite ← (CFacts.exists_iff C (compat_bool_2 A1 Ax t0)).
          ∃ (t0, t1). split. apply H1.
          apply andb_true_intro. split. apply andb_true_intro. split.
            apply → mem_iff. apply H6. apply → mem_iff. apply H7.
          apply ← beq_pkg_eq. reflexivity.
```

Next case: $t_0 \in A_x$ and $t_1 \in A_1$. In this case, t_1 would be in $K(A_1, A_x)$.

```
          destruct (union_1 H3).
              apply H5. unfold packages_to_remove. rewrite (filter_iff (union A1 Ax) t1
  (compat_bool_3 C A1 Ax)).
              split. apply H3. rewrite ← (CFacts.exists_iff C (compat_bool_2 A1 Ax t1)).
              ∃ (t1, t0). split. apply conflicts_sym. apply H1.
              apply andb_true_intro. split. apply andb_true_intro. split.
                 apply → mem_iff. apply H7. apply → mem_iff. apply H6.
              apply ← beq_pkg_eq. reflexivity.
```

And finally, t_0 and t_1 both in A_x. This is impossible, because A_x is peaceful.

```
              apply (HAx_P (t0, t1)). apply H1. split. apply H6. apply H7.
```

It just remains to prove that $\{x\}$ and a_1 are co-installable. First x:

```
          elim (Hsep x). intros. ∃ x0. destruct H1. destruct H2. split.
              apply H1. split. simpl. unfold Subset. intros. apply In_1 with x. apply
  singleton_1. apply H4. apply H2.
              apply H3.
          elim (H0 x). intros. apply H2. left. reflexivity.
```

And finally, a_1.

```
          elim (IHa1 Ha1). intros. destruct H1. destruct H2.
          ∃ x0. split. apply H1. split. apply H2. apply H3.
          intros. apply (Hsep x0). elim (H0 x0). intros. apply H3. right. apply H1.
          unfold only_triangles. unfold ConflictSet.For_all. intros.
          apply Htri. apply H1. unfold concerned in H2. destruct x0. destruct H2.
          unfold concerned. split.
          apply (cone_of_subset_is_subset _ _ R (exist (fun v ⇒ v [≤] R) a1 Ha1)).
              simpl. unfold Subset. intros. destruct (H0 a). apply H6. right. apply H4.
              apply H2.
          apply (cone_of_subset_is_subset _ _ R (exist (fun v ⇒ v [≤] R) a1 Ha1)).
              simpl. unfold Subset. intros. destruct (H0 a). apply H6. right. apply H4.
              apply H3.
          intro. apply Hcc. destruct H1. ∃ x0. destruct H1. split.
              apply H1.
              destruct x0. destruct H2.
                 left. destruct (H0 t0). apply H4. right. apply H2.
                 right. destruct (H0 t1). apply H4. right. apply H2.
```

This concludes the proof.

Experimentation and validation 7

. . . and searched the Scriptures daily, whether those things were true.
— Acts 17:11

In the previous chapters, one of the justifications we have proposed for our work has been that it offers distribution editors more insight into the structure of their distribution, and therefore better ways of finding errors and, in general, maintaining the quality of their product.

In this chapter, we shall discuss the results of various practical experiments we have executed during the MANCOOSI project, and show the information that can be gathered from these experiments.

The first part of the chapter will be devoted to a discussion of the time involved in the computation of the notions discussed in previous chapters. In chapter 4, we have discussed the theoretical complexity of the algorithms used, but in every case, the actual run times are far below these theoretical maxima. In addition to a discussion of this phenomenon, we also present a quantitative assessment of the results of the calculations.

The second part of the chapter contains an analysis of the results themselves: patterns and other interesting features that have been uncovered by studying the results.

For the experiments, we have made use of the latest Debian stable repository amd64 architecture; the latest Mandriva repository, 2010.1; and the newest written by Çagdas Bozman [Boz10].

7.1 Repositories

One fairly basic question one might ask oneself is how the number of packages and dependencies evolve over time, especially in relation to each other. In order to answer this question, consider figure 7.1, which shows the evolution of the number of packages and dependencies in the Debian `unstable` distribution from January 2007 to mid-June 2010.

The 'number of dependencies' in this graph is obtained by taking all direct dependency relationships in the package; that is, for every pair of packages (p, q) such that $p \rightarrow q$ (i.e. q is mentioned in the dependency specification of p), one is added to the total.

The first thing we can see is that both the number of packages (the red line, scaled on the left side of the graph) and the number of dependencies (the green line, scaled on the right of the graph) grow in a linear fashion over time, with one notable exception: from mid-November 2007 to August 2008, the number of packages increases, but the number of dependencies decreases. This is probably due to a change of packaging in packages that use the `pkg-config` framework.
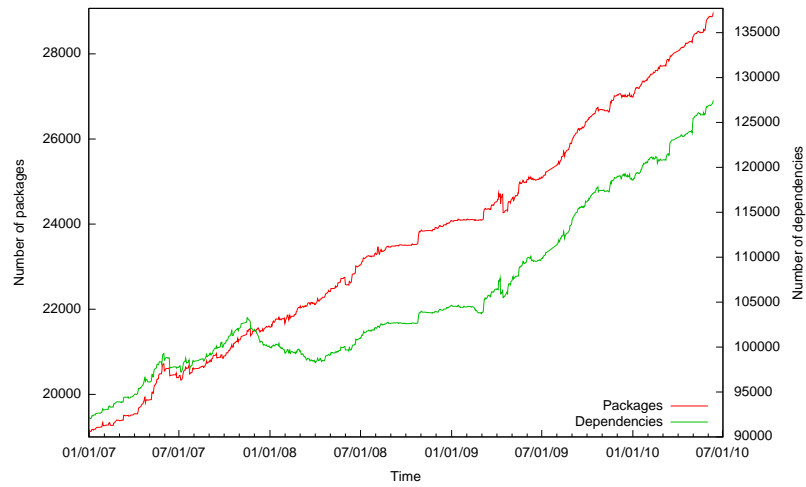
Figure 7.1: Packages and dependencies in Debian `unstable`

Let us note in passing that the scale on both Y axes is equal, so that apart from the aforementioned period, the growth rate for both packages and dependencies really is equal.

This is even easier to see in a plot of the average number of (direct) dependencies per package against the time, as in figure 7.2:
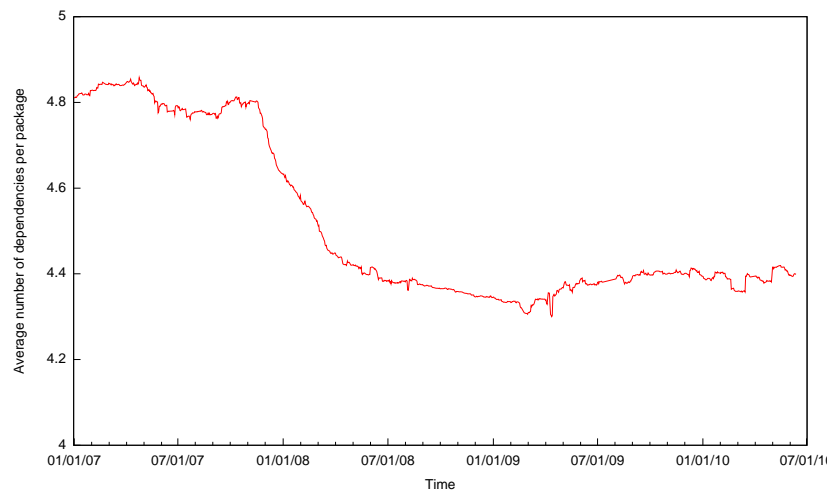

Figure 7.2: Average number of dependencies per package in `unstable`

Before mid-November 2007, there are on average around 4.8 dependencies per package; this drops to 4.4 in three months, where it remains until the present time.

7.2 Run time of the installability algorithm

The basic algorithm presented in the previous chapters was the installability problem; more specifically, checking a distribution for packages that are not installable under any circumstances ('broken'). We have noted already that while this problem is NP-complete and therefore theoretically exponential in complexity, the actual instances encountered when checking distributions can be solved very quickly. In this section, we shall present data to corroborate this.

7.2.1 Installability for a single distribution

To find out the actual run time of the algorithm that checks for broken packages, we shall take the contents of the Debian and Mandriva distributions over time, and check the run times. For Debian, we start with version 2.0, since previous distributions were too small to give good results.

The results for Debian are in the following table[1].

Distribution	Year	Packages	Broken	Solver time
2.0	1998	1524	1	0.03
2.1	1999	2269	7	0.06
2.2	2000	3889	12	0.21
3.1	2005	15195	2	4.15
4.0	2007	18052	5	3.37
5.0	2009	22311	0	4.74
5.0.6	2010	22000	9	4.15

distributions used: debian/i386 main

For Mandriva, the results are as follows:

Distribution	Packages	Broken	Solver time
2007.0	5123	53	0.77
2007.1	5517	39	0.61
2008.0	6083	11	0.75
2008.1	6410	28	0.98
2009.0	6860	69	0.73
2009.1	6936	29	0.70
2010.0	7340	20	1.46
2010.1	7566	61	2.04

distributions used: mandriva main/i386

These stable distributions give us some data points, but not enough to give a clear indication of a relation between distribution size and run time.

To get a better indication of this relations, We have run the algorithm on every Debian `stable`, `testing` and `unstable` distribution (for the `amd64` architecture) of the first day of the month from May 2007 through June 2010. The results are shown in figure 7.4[2].

---

[1]All run times in this chapter and the next were obtained by running the tools on a 4-core, 3 GHz, AMD Athlon II, with 4 Gb of memory and running FreeBSD 8.1.

[2]All figures from this section can be found at the end of the chapter.

The curve is mostly linear, though there are significant outliers. This is because there can be great differences between two distributions that are similar in size: as an example, the `unstable` distribution for June 1st, 2008, has almost the same size as the `testing` for December 1st, 2008, but the `testing` distribution takes 20% longer to solve. This can be attributed to the fact that there are 5044 broken packages in the `unstable` distribution: many of these broken packages are broken because of the same reason, which greatly speeds up the work of the SAT solver.

7.2.2   Installability for multiple distributions

What happens if one takes two different distributions (stable and testing, or testing and unstable) and check them for broken packages? Looking at figure 7.5, we can see that the progression is still linear, but that the constant becomes much higher.

To explain this, let us note first that if two distributions are added together, in many cases, this union contains two different versions of the same package. So, many dependencies that before had only one solution now have two solutions; and to add insult to injury, these two solutions are in conflict with each other.

Let us give an example. In Debian 5.0.4, from January 2010, there is one version of the package `libc6`, to wit `2.7-18lenny1`. Any dependency on `libc6` will therefore have exactly one way to resolve it.

However, in a Debian testing from May 2010, there is another version of `libc6`: `libc6-2.10.2-6`. Since the dependency on `libc6` is usually specified as `libc6 (>= 2.7-1)`, this second version of `libc6` also satisfies the dependencies, of which there are some 10 000. This enormously complicates the task of the SAT solver.

And when `libc6` from the Debian unstable of the same date is added, there is another version: `libc6-2.10.2-7`, which triples the problem.

Another interesting observation from figure 7.5 is that there are three distinct lines to be seen. The first line corresponds to the distributions from May 2007 up to and including February 2009. On February 14, 2009, Debian 5.0 was released, so that the up-to-then testing became stable. This resulted in a decrease in the number of packages (the combination stable and testing dropped from around 38 000 packages to less than 28 000). Given that on that date the stable distribution contained some 22 000 packages, there were a lot less packages with multiple available versions (as explained in the previous paragraph), which explains the reduction in solver time.

The second line, therefore, represents the packages from March 2009 up to and including November 2009. There is a smaller drop here, which coincides with an upgrade of the `libc` package (reducing the number of `libc` packages available from two to one). We have already seen that this package is depended on by a great many packages in the distribution, so this explains the effect (every package that depends on `libc` no longer has to make the choice between two versions, but can simply install the single available version).

This becomes clearer in a 3d plot of the number of packages versus the solver time versus the date, as in figure 7.6. The actual plot in this graph is the red line; the two other black lines are just projections of this line on the back and bottom plane, added for clarity.

Following the red line shows that up until February 2009, there is a steady increase of both the number of packages, as well as the solver time; this is the expected behaviour which also occurs when checking single distributions. Then, there is a sudden drop which brings us back to the same number of packages (and solver time) as in February 2007. This is the release of Debian 5.0 mentioned before.

The second phenomenon mentioned, the change in `libc`, is less obvious in this graph; it is the small drop in November 2009. Looking at the two projections, we can see that the number of packages does not change (the line on the back plane), but that there is a drop in solver time (the bottom plane).

Indeed, the `libc` package itself is only one package, but there is a difference in solver time, because all packages that depend on `libc6` can be solved more easily when there is only one available version.

Looking at figure 7.7, where the numbers for the union of stable, testing and unstable have been added, we can see that the same phenomenon is observable there.

A 3d version of this plot, with only the line for stable, testing and unstable shown, is in figure 7.8.

7.3   Run time of the strong dependency algorithm

Let us continue the run time analysis with the algorithms that are based on this installability algorithm, starting with the algorithm used for computing the strong dependency graph of a distribution. In chapter 4, we noted that the problem of determining whether two packages strongly depend on each other is co-NP-complete; we shall show here that the actual time needed to produce a strong dependency graph of an entire distribution (using the algorithms from chapter 4) remains reasonable.

The following table shows the run time of algorithm 5, the most highly optimised algorithm shown to compute the strong dependency graph.

In this table, V and E are the vertices and edges in the strong dependency graph (the transitive version); and Time is the time needed to construct this graph (in seconds).

Distribution	V	E	Time
stable (5.0.6)	22 000	729 977	98.78
stable + testing	47 434	244 899	3371.83
stable + testing + unstable	50 533	239 419	4160.16

distributions used: debian/amd64 main

The first observation to be made here is that for a single distribution, the strong dependency graph can be computed in a matter of minutes. While this is not enough to run this in real time, it is more than fast enough to run as part of a daily analysis for the benefit of distribution editors.

We can also observe is that if the algorithm is run on multiple distributions together, not only does the run time increase dramatically, but the number of edges in the strong dependency graph also *decreases*.

This is due to the fact that when considering multiple distributions at the same time, there are multiple versions of the same package, which will provide

more ways of satisfying dependencies. Since a dependency that can be satisfied by multiple packages is not a strong dependency, the number of strong dependencies decreases.

The increase in run time is due to this same effect. First, there will be less conjunctive dependencies, so the SAT solver will have to be invoked more often to determine whether a dependency is a strong dependency or not. In addition, as already noted in the previous paragraph, having multiple ways to satisfy a dependency complicates the task of the SAT solver.

An example of this is figure 7.3. In the repository 7.3a, there is a direct conjunctive dependency from `alpha` on `bravo`, and therefore a strong dependency. If the actual specification of this dependency is just `bravo`, however, without any version numbers, this dependency is no longer conjunctive when another version of `bravo` is added, as in repository 7.3b. Since it is now possible to install either version of `bravo` in order to satisfy the dependency of `alpha`, the strong dependency disappears.

(a) One version   (b) Two versions
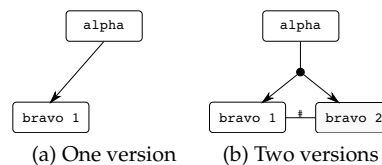
Figure 7.3: Strong dependencies and multiple versions

7.4   The dominator graph

The algorithm for computing the dominator graph uses the strong dependency graph algorithm. As we have noted before, there are two different methods of computing the dominator graph: a 'classical' method presented in chapter 4 and a 'Tarjan' method which takes advantage of the fact that dominators for dependency graphs are equivalent to the notion of dominators as known from control flow graphs (see theorem 3.19) to use the Lengauer-Tarjan algorithm [LT79] to compute the dominator graph.

In the first part of this section, we shall discuss the run time of these two algorithms.

In the second part, we shall discuss interesting patterns that can be found in the dominator graphs thus computed.

7.4.1   Run time of dominator graph calculation

Let us start with the first algorithm (algorithm 7): the classical method that follows the transitive strong dependency graph and finds out if there are dominators (see chapter 4 for a more extensive presentation). This algorithm is theoretically very slow ($O(|V|^2)$ for a completely connected graph), but in practice its run time remains reasonable, as can be seen in the following table. V and E are the number of vertices and edges in the strong dependency graph; Dom

denotes the time needed to generate the dominance graph, and RelDom5 the time needed to generate the relative dominance graph (up to 5 percent; see definition 3.10). This does not comprise the time needed for creating the strong dependency graph, since it can be found in the table in the previous section.

Distribution	V	E	Dom	RelDom5
stable (5.0.6)	22 000	792 977	6974.43	7051.04
stable + testing	47 434	244 899	467.27	395.35
stable + testing + unstable	50 533	239 419	368.50	371.38

distributions used: debian/amd64 main

Then there is the Tarjan algorithm (algorithm 9). This algorithm uses the strong dependency graph as a starting point, in a modified form, where cycles have been replaced by a single vertex and that has been transitively reduced (which explains the reduced number of vertices and edges, in comparison to previous graphs).

As noted in chapter 4, the theoretical complexity of computing the transitive reduction of a graph is $O(|V|^3)$. In practice, computing the transitive reduction of a strong dependency graph never takes more than a few seconds. This can be explained by the nature of the graph, which, though transitive and therefore having many more edges than vertices, is far from being completely connected; the strong dependency graph of Debian 5.0.6, for example, contains only 0.2% of the number of possible edges.

In the table below, the V and E columns contain, respectively, the number of vertices and the number of edges of the strong dependency graph (transitively reduced and with cycles replaced by a single node). The TG column contains the time needed to generate this reduced strong dependency graph, and the TD column is the run time of the Tarjan algorithm. The total run time is the entire run time of the algorithm, which includes parsing of the distribution file (and therefore is higher than the sum of TG and TD). All these times are in seconds.

Distribution	V	E	TG	TD	Total
stable (5.0.6)	21 940	43 412	50.06	103.69	160.15
stable + testing	47 373	41 796	451.53	334.73	801.26
stable + testing + unstable	50 465	42 854	508.52	375.08	904.63

distributions used: debian/amd64 main

7.4.2 Characteristics of the dominator graph

Considering the structure of the dominator graph, let us note that it consists of a large amount of independent clusters. Many patterns are noticeable here. For example, the largest cluster in a recent `testing` distribution, consisting of 47 nodes, is the one shown in figure 7.9[3].

The node in the centre is `cairo-dock-plugins`, and it dominates all the nodes at the edge of the graph.

---

[3]All cluster images from this subsection can be found at the end of the chapter

This cluster models a package (`cairo-dock-plugins`), with a lot of dependencies that are only reachable via this package (this is because of the dominator relationship). This could be interesting for analysis purposes, since basically this entire cluster is one package that can be installed in part (only the needed plugins)—it is certain that the plugins will not be needed by someone else, so removing them will never be a problem.

In figure 7.10, there is an example of a smaller cluster, focusing on the `tzdata-java` package.

This pattern is interesting as well: it shows us three packages that mutually dominate each other: `openjdk6-jre-lib`, `openjdk16-jre-headleass` and `ca-certificates-java`; these three packages all dominate `tzdata-java`.

The first three packages will always be installed together, so it might be interesting to consider merging them into a single package—of course, there could be external factors to prevent this, such as the fact that they have different maintainers, or licensing issues.

Finally, as seen in figure 7.11, much more complex patterns exist as well.

What can be seen here is a combination of several clusters, dominated by the `kdepim-runtime` package.

Sometimes a cluster seems more complicated at first sight than it actually is. Consider the cluster around `mono-devel`, in figure 7.12:

This looks complicated, but actually many of the edges shown here are transitive edges. When they are removed, a more recognisable pattern emerges of a central package that dominates a great number of other packages, like already seen in figure 7.9.

7.5 Run time of the strong conflict algorithm

The algorithm used for computing strong conflicts, as presented in chapter 4, theoretically has a very high complexity. As shown in that chapter, many optimisations are possible to reduce the practical run time to something acceptable.

In the first part of this section, we shall discuss the actual time needed to compute the strong conflicts in a distribution. In the second part, we shall present a few significant examples from the Debian distribution.

In the following table, there is data about the running of the strong conflicts algorithm. As an example, for Debian, there are 1003 explicit conflicts (E), of which there are 940 left after removal of triangles (T). Running the algorithm results in some 10 million potential strong conflicts (P; i.e. pairs connected to an explicit conflict by a dependency path that contains at least one disjunctive dependency). There are 6600 conjunctive strong conflicts (CSC; i.e. pairs connected to an explicit conflict by a conjunctive dependency path), and finally, of the potential strong conflicts, 1714 were actual strong conflicts, giving 8314 total strong conflicts (SC).

Distribution	Packages	Trimmed	E	T	P	CSC	SC	Time
debian 5.0.4	22 299	22 295	1 003	940	10 071 609	6 600	8 314	1309s
mandriva 2010.1	7 566	7 505	51	50	2 542	248	282	80s
eclipse-helios	4 027	3 996	25	22	883	33	33	13s

When we look at the numbers for Debian first, we can see that the larger part of the strong conflicts are found directly using only conjunctive dependencies. About ten million possible strong conflicts (that is, pairs of packages that have a path to a conflict that includes at least one disjunctive dependency; see also theorem 3.25) are found, but only a very small part of these possibilities actually are strong conflicts.

Note also that while the triangle optimisation only removes 63 conflicts, one of these is the conflict between the packages `debconf-i18n` and `debconf-english`. Since these two packages are (through `debconf`) heavily depended on, the removal of this conflict reduces the number of candidates by a huge amount.

When we look at the specific conflicts, it turns out that almost 40 percent of the strong conflict candidates have the conflict between `libgamin0` and `libfam0` as their root. This is not unduly surprising, since the gamin and fam libraries are well used (as can be verified when looking at the number of packages that strongly depend on them).

When we look at the list of the 10 packages with the highest inclusion set, as shown in table 7.1, we can see that the size of the exclusion set is an excellent method to spot problematic packages.

Exclusion set	Package	Explicit conflicts	Explicit dependencies	Closure size	Closure height
2362	ppmtofb	2	3	6	4
127	libgd2-noxpm	2	5	8	4
127	libgd2-noxpm-dev	4	6	15	5
107	heimdal-dev	2	8	121	10
71	dtc-toaster	0	11	429	9
71	dtc-postfix-courier	2	22	348	8
69	citadel-suite	0	5	133	9
69	citadel-mta	1	6	123	9
65	xmail	4	6	105	8
63	apache2-mpm-worker	2	5	123	10

Table 7.1: Top 10 of packages with large exclusion sets in Debian 5.0.6

7.5.1 Significant examples of strong conflicts in Debian

Let us look at the packages in the Debian top 20 for some more detail, to see what explanations can be found for their large exclusion sets.

To begin with, the case of `ppmtofb` has already been shown in chapter 3: a dependency on an obsolete component.

Insufficient precision in the metadata

When we look at the two next packages in the top 20, `libgd2-noxpm` and `libgd2-noxpm-dev`, we can immediately see that these packages seem very similar. Let us look at the presentation given by the tool:

```
127 libgd2-noxpm-dev:
127 (libgd2-xpm <-> libgd2-noxpm)
* zoph (conjunctive)
(...)
127 libgd2-noxpm:
127 (libgd2-xpm <-> libgd2-noxpm)
* zoph (conjunctive)
(...)
```

All strong conflicts of *both* packages are caused by *one* explicit conflict. Looking at the metadata, this conflict seems justified: it is reasonable to assume that `libgd2-xpm` and `libgd2-noxpm` are two versions of the same library and therefore should not be installed together (and indeed, by studying the metadata further, this turns out to be the case).

In fact, both packages contain the same library, but `libgd2-noxpm` is a base version, whereas `libgd2-xpm` adds support for the XPM (X PixMap) image format. It seems reasonable, therefore, to say that the functionality of `libgd2-xpm` is a superset of the functionality of `libgd2-noxpm`.

The two packages do not actually conflict with each other; it is true that they cannot be installed together, but it is always possible to resolve the conflict by removing `libgd2-noxpm` and installing `libgd2-xpm`, without losing any functionality.

This could be reflected in the metadata by adding a `Replaces` field or by having `libgd2-xpm` provide `libgd2-noxpm`, thus removing the strong conflicts.

For the next package, `heimdal-dev`, 106 of its 107 strong conflicts are caused by one explicit conflict, the one between `libkrb5-dev` and `heimdal-dev`. This falls in the same category as the previous case: `heimdal` aims to be a compatible reimplementation of `libkrb5` (the Kerberos security system). But in that case, should there be a conflict between the two? Apparently, if both libraries are compatible, it does not matter which one is installed. This situation might be resolved by adding a virtual package `kerberos-dev`, which is provided both by `libkrb5-dev` and `heimdal-dev`.

Justified strong conflicts

Not all strong conflicts are problematic, however. For example, next in the list is `dtc-toaster`, which has 71 strong conflicts. Here the pattern is different: there is not one explicit conflict that is the root cause of all these strong conflicts, but many different ones (the most important one being the one between `courier-mta` and `postfix`, which causes a strong conflict between `dtc-toaster` and 29 other packages): in total there are 24 explicit conflicts at the root of this exclusion set.

This exclusion set therefore seems to be justified. Given that 71 packages is only 0.3% of the total distribution size, such an exclusion set does not seem to pose a very large problem anyway.

Note that all the solutions given here are propositions: it might very well be that there are technical difficulties that preclude these solutions. The important lesson is that with the presentation given here, it is very simple to find problematic packages and note the root cause that leads to their having such large exclusion sets.

7.5.2 Strong conflicts in Mandriva and Eclipse

Then, let us look at to the results for Mandriva, shown in table 7.2. Here, there are far fewer explicit conflicts (even speaking relatively, taking into account the fact that the Mandriva distribution is smaller than Debian), and consequently there are less strong conflicts as well.

This is a result of Mandriva (and RPM) design policy: in general, the idea in Mandriva is to alter packages that potentially conflict to have them exist alongside each other, which is not the case in Debian.

Exclusion set	Package	Explicit conflicts	Explicit dependencies	Closure size	Closure height
32	vsftpd	4	25	62	8
30	pure-ftpd	5	24	47	6
30	pure-ftpd-anonymous	0	1	48	7
30	pure-ftpd-anon-upload	0	1	48	7
18	lib64db4.7-static-devel	11	1	58	10
18	lib64db4.7-devel	11	5	57	9
18	lib64db4.6-static-devel	9	1	62	10
18	lib64db4.6-devel	9	6	61	9
18	lib64db4.2-static-devel	6	1	62	10
18	lib64db4.2-devel	6	6	61	9

Table 7.2: Top 10 of packages with large exclusion sets in Mandriva 2010.1

When we look at the package from Mandriva with the most strong conflicts, we can immediately see that proportionally, there are not as many as for Debian: 32 on 7566, less than 0.5%.

In fact, the four top packages in the strong conflict list are related: the strong conflicts involving vsftpd, pure-ftpd, pure-ftpd-anonymous and pure-ftpd-anon-upload all involve two explicit conflicts: one between proftpd and vsftpd and one between proftpd and pure-ftpd.

Since all these packages involve FTP software, here again some adjustment in the metadata might eliminate these conflicts, like we have shown for the libgd2 packages above; but this of course depends on the distribution and the specifics of the package. At any rate, we can see again that using the root conflict, it is easy to identify the underlying cause of those large exclusion sets.

For Eclipse, there are no packages that have large exclusion sets (the largest exclusion set found has three elements); besides, every strong conflict is due to a self-conflict (i.e. an explicit conflict of a package with itself, which in fact means that the singleton flag is set, as explained in chapter 2).

This is due to the peculiarities of the Eclipse format: there are no explicit conflicts between plugins; the Eclipse quality assurance process insures that plugins from the official distribution are always usable together.

7.5.3 Daily generation of data

All of the algorithms discussed in this chapter are intended to provide data to distribution editors, so that they can in turn improve the distribution.

In order to make this easy, on the MANCOOSI site, we have put a system in place that generates the dominator graphs of Mandriva and Debian (`testing` and `unstable`; not `stable`, since it does not change every day) on a daily basis.

This daily generation also allows distribution editors to check the evolution of their distribution over time; data is available from April 2009 onwards.

Since, as noted above, the dominators graph is highly clustered, there also is a break-down of the graph into its component clusters. This can be used to monitor changes in dependencies; for example, to check if packages move between clusters, if clusters merge or if they disappear. This might be indicative of changes in the distribution.
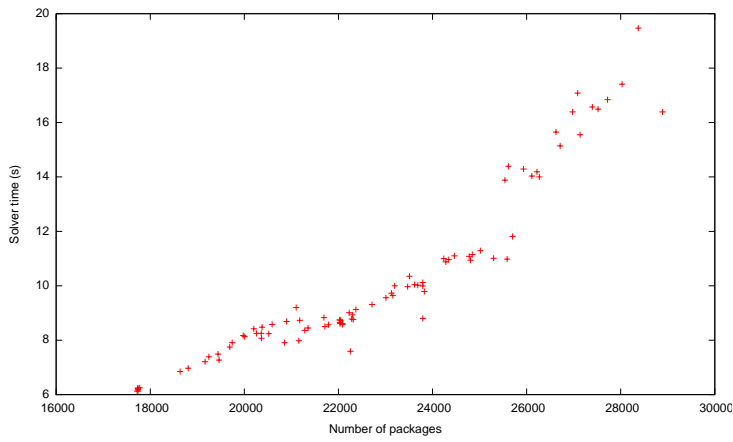
Figure 7.4: Solver times for one Debian distribution

distributions used: debian main/amd64
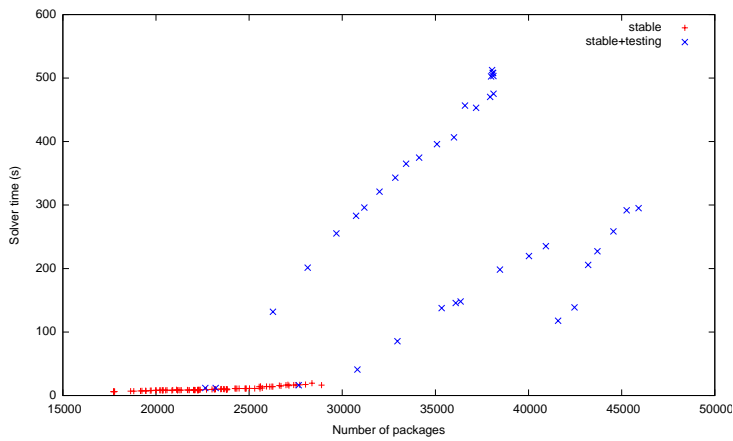
Figure 7.5: Solver times for stable and testing
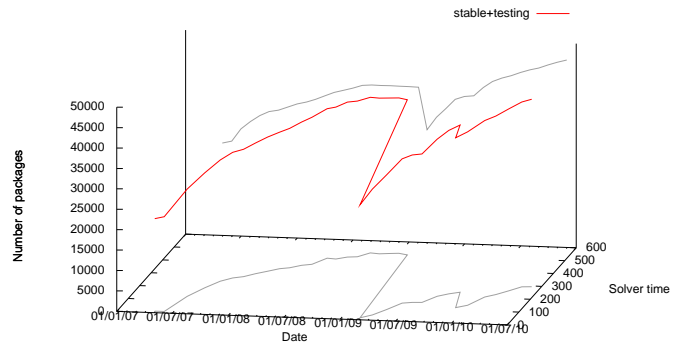
Figure 7.6: Solver times for stable and testing, in 3D
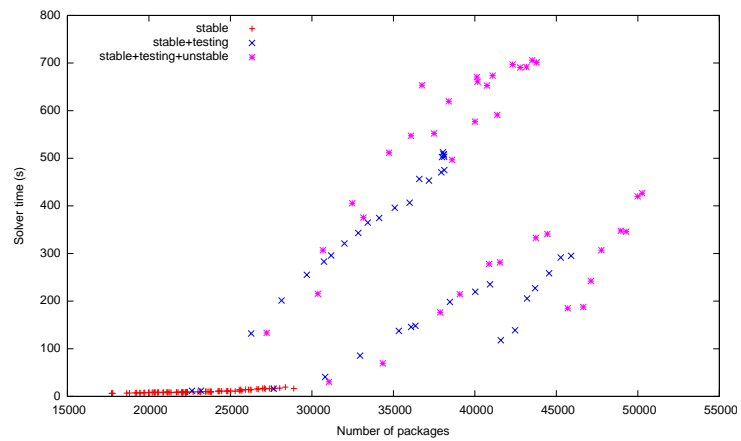
Figure 7.7: Solver times for three distributions
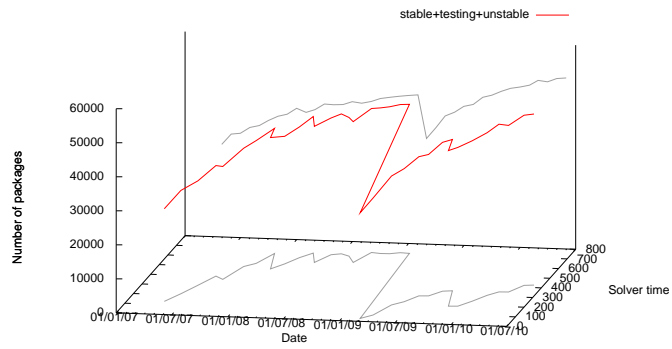
Figure 7.8: Solver times for three distributions
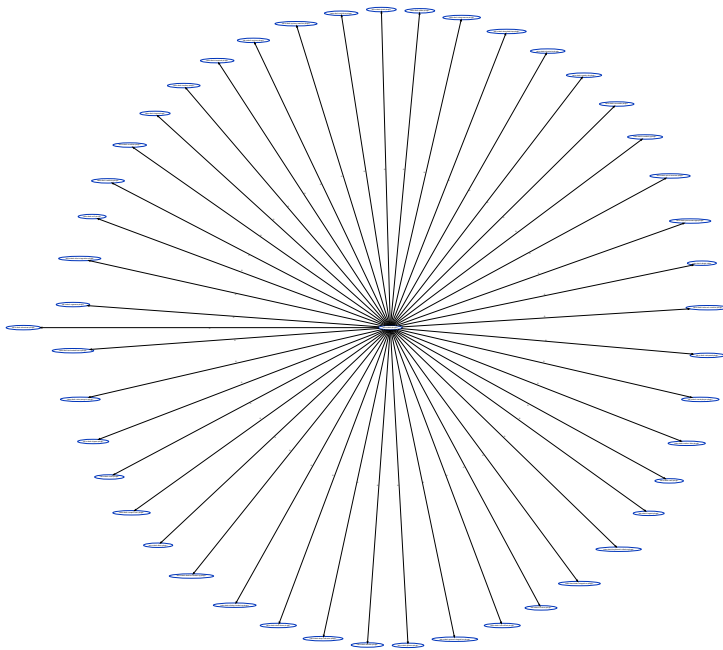
Figure 7.9: Cluster around `cairo-dock-plugins`

Figure 7.10: Cluster around `tzdata-java`

Figure 7.11: Cluster around `kdepim-runtime`
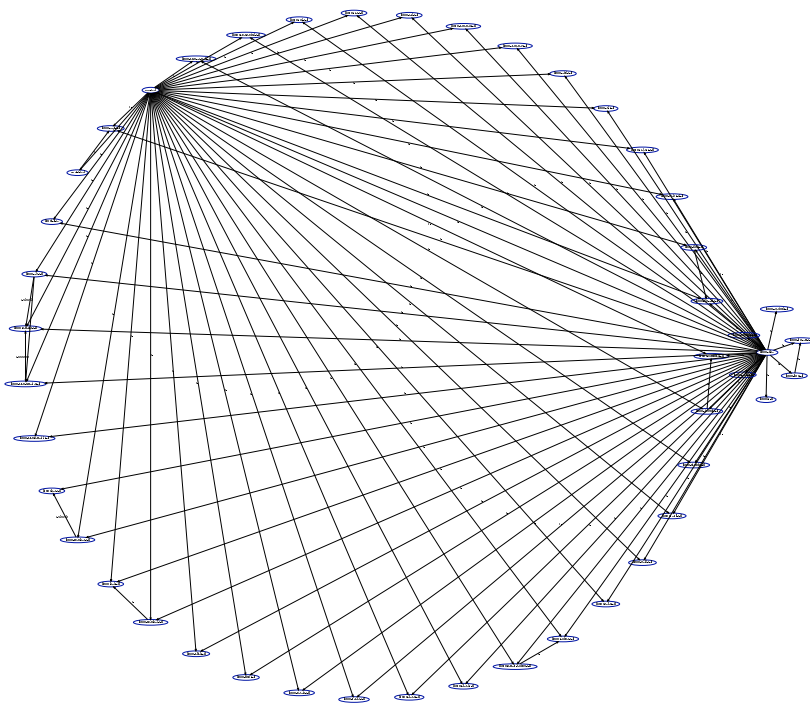
Figure 7.12: Cluster around `mono-devel`

Graph properties of distributions 8

Qualsiasi dato diventa importante se è connesso a un altro. La connessione cambia la prospettiva.

— UMBERTO ECO, *Il pendolo di Foucault*

The relationships between components can be used to compute relevant quality measures, for example in order to identify particularly fragile components [DML+04, KAJH99, Liv05]. This chapter is devoted to a discussion of this idea; more specifically, we shall discuss the properties of distributions when seen as a graph.

It has been observed that in networks of objects generated by human activity it is often the case that the number of hops needed to reach a vertex from another vertex is relatively small. This is called the *small world* property, first described by Stanley Milgram in [Mil67] (the famous concept of 'six degrees of separation'), and can be found in the graph of the Internet [CMP00], the graph of the Web [BA99], and many other types of complex networks [AB02]. As Barabasi says in his founding paper on small world properties, it seems that these large networks are "governed by robust self-organizing phenomena that go beyond the particulars of the individual systems".

It is quite natural to ask the question whether free software distributions, whose properties are being discussed in this thesis, also exhibit such scale-free behaviour when considered as a graph. This is the question we are going to answer in this chapter, after recalling the basic notions about the 'small world' properties of a graph.

8.1 Small world properties

Formally, a network is deemed small-world if it satisfies two properties [WS98]:

1. The clustering coefficient (the chance that two neighbours of a vertex are connected) is significantly higher than that of random networks.

2. On average, the path length between two vertices is low (on the same order as that of random networks);

Let us start with a more formal definition of these two concepts, the clustering coefficient and the path length. Assume that there is a directed graph G, with V the set of its vertices and $E \subseteq V \times V$ the set of its edges (if $(p,q) \in E$, then there is an edge from p to q).

Definition 8.1 (Clustering coeffcient)
Given a vertex v, the clustering coefficient $\mathrm{cc}(v)$ *is defined as:*

$$\mathrm{cc}(v) = \begin{cases} 0, & \text{if } \mathrm{succ}(v) = \emptyset \\ 1, & \text{if } \mathrm{succ}(v) = \{p\} \text{ for some } p \\ \frac{|\{(p,p') \in E | p,p' \in \mathrm{succ}(v) \wedge p \neq p'\}|}{\frac{1}{2} * |\mathrm{succ}(v)| * (|\mathrm{succ}(v)| - 1)} & \text{otherwise} \end{cases}$$

The clustering coefficient $\mathrm{cc}(G)$ *of a graph G is the average clustering coefficient of all of its vertices.*

Then the average path length. First, the path length $P(v, v')$ is defined as the length of the shortest path between v and v'; if v and v' are not connected, then $P(v, v') = \infty$.

Definition 8.2 (Average path length)
The average path length $\mathrm{apl}(v)$ *of a vertex v is defined as:*

$$\mathrm{apl}(v) = \begin{cases} 0, & \text{if } \mathrm{succ}(v) = \emptyset \\ \frac{\sum \{P(v,v') | P(v,v') \neq \infty\}}{|\{P(v,v') | P(v,v') \neq \infty\}|}, & \text{otherwise} \end{cases}$$

Again, the average path length $\mathrm{apl}(G)$ *of a graph G is the average of the average path lengths of all of its vertices.*

In a small world network, these two properties result in a network that consists of many clusters (or near-clusters), whose central nodes are connected. This makes for the short average path lengths mentioned above.

These hubs can be recognised by looking at the degrees of the different nodes in the graph. There are a few hubs—nodes that have many connections—and many nodes that have very few connections; thus, the degree distribution conforms to the well-known Pareto principle, also known as the 80/20-law.

When looking at a plot of the degrees of a Debian `stable` distribution graph, as presented in figure 8.1, this pattern can be recognised. In this graph, the degree is presented on the Y axis, and the number of nodes that has this particular degree is presented on the X axis. We can see easily that there are many nodes with a low degree, and few nodes with a high degree.

This particular structure makes such a graph very resistant to random attacks [AJB00]; if a random node is removed, the chance is very small that the average path length will increase: the only way the path length can be increased for many nodes is by removing a hub node, and as shown above, there are few of these.

By the same argument, however, a small world graph is very vulnerable to *directed* attacks: by deleting only a few specific nodes (the hub nodes), the entire network can be paralysed.
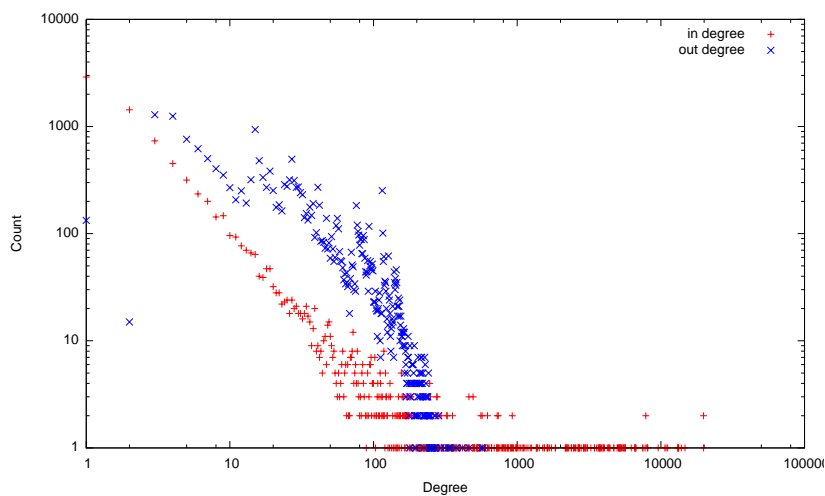
Figure 8.1: Distribution of degrees in Debian `stable` (method 3)

8.2 Distributions as small world networks

The argument presented above makes sense from a distribution principle as well: there are many packages in a distribution (all sorts of applications) that, if broken or removed (which can be seen as an 'attack'), will not have much impact on packages other than themselves. The remark about directed attacks holds as well: there are a few packages (libraries such as `libc`, for example, or script interpreters such as `perl` or `python`) that, in case of bugs or other problems, will have a major impact on the distribution.

In this section, we shall discuss the application of the small world concept to F/OSS distributions. Such distributions exhibit important differences with other objects usually studied as small world networks, but as we shall show, it is still possible to apply many of the concepts from small world networks to distributions.

Most network models so far have simply ignored the *directed*ness of the network, treating all edges as if they were symmetric. However, there are networks where this is not the case; for example, in the World Wide Web, there is a difference in distribution between ingoing and outgoing edges [BKM+00] (both follow a power law, but the distribution is different).

In the definitions for clustering coefficient and path lengths mentioned above, the directedness of the graph has already been considered, as well as in the degree distribution (here, too, there is a difference in distribution between incoming and outgoing edges, though both obey a power law).

There is a second difference, however: unlike the Internet, the Web, or social networks, a distribution graph contains different kinds of edges: while a *dependency* edge may well be considered as a connection in the same sense as in a social network, a *conflict* edge has a very different meaning. Also, dependencies are *directed* edges, while conflicts, which specify a symmetrical relationship, are *undirected* ones.

Therefore, it is not at all obvious what graph one should use to check whether

the small world properties apply. At least three main possibilities for graph generation methods can be identified:

1. Treat *all* edges equally, irrespective of their significance (dependency, recommendation, conflict, ...). More formally, there is an edge between packages p and q if and only if q is mentioned in one of the relationships of p (notice that this notion totally misrepresents the semantics of a conflict, which is symmetric)

2. Use the dependency closure; there is an edge between p and q if and only if q is a member of $\Delta(p)$. This creates an edge between p and all possible dependencies of p.

3. Use strong dependencies: there is an edge between p and q if and only if $p \Rightarrow q$[1].

All these possibilities have different properties. If the idea is for an edge from p to q to represent the fact that q is needed to install p, then using the dependency closure will result in an over-estimation (not every package in the dependency closure is installed every time, but packages that are not in the dependency closure are guaranteed not to be needed), whereas using strong dependencies will result in an under-estimation—a strong dependency is guaranteed to be installed, but usually other packages will be installed as well to satisfy alternative dependencies.

There has been some prior research on the scale-free behaviour of Debian networks; in [LW04], it is claimed that Debian distributions show small-world characteristics.

It is difficult, however, to attach conclusions to this claim, because the methodology used is not discussed in any detail. The numbers shown in the paper suggests that the authors have added an edge between two packages if there was a dependency (conjunctive or disjunctive) between two packages, which is akin to our method 1.

The problem with this approach is that such a relationship can mean very different things. First, that a package depends on another does not necessarily mean that this other package is actually going to be installed; if the dependency is disjunctive, another package might be chosen to satisfy the dependency. And second, a package can be installed even if it is not linked by a direct dependency: see chapter 3 on strong dependencies for more information on this.

Another paper [NNR09], also claims to have found small-world characteristics in the different stable Debian distributions.

Again, it is not clear which methodology has been used, but the numbers suggest that for the dependency relationships, the same approach in graph generation has been used as in [LW04].

There are two differences in approach between the two papers, though: in [NNR09], a distinction is made between the graph of dependency relationships and the graph of conflict relationships; and the directedness of the graph is taken into account insofar as the in and out degrees are treated separately.

---

[1]This is the method used to create the degree distribution graphs presented in the previous section.

Furthermore, it is argued in this paper that the small-world model (as demonstrated by the degree distribution) does not hold for packages with either very few or very many dependencies (the 'saturation effect').

This is a puzzling claim to make, because the whole point of the small-world phenomenon is to distinguish between those two types of packages. Our results tend to show that FLOSS distributions conform very well to the small-world model: there are few packages with many dependencies, and many packages with few dependencies.

In [MSSvK08] and [NNR09], the validity of Zipf's law [Zip49] for Debian distributions has been demonstrated. However, again, in these papers the sort of graph used for the calculations is not mentioned.

It turns out that with respect to small-world networks, it does not matter which representation is chosen: all three graphs have the small world property, though the numbers differ.

Method	V	E	CC	APL	Comp	CpAvg	LComp
1	22 000	177 805	0.46	8.27	338	65.09	21 655
2	22 000	1 726 931	0.30	0.91	1 427	15.42	20 519
3	22 000	792 977	0.29	0.91	1 484	14.82	20 445

distribution used: debian/amd64 5.0.6 stable

In this table, the three methods are represented. First V and E, the number of vertices and edges in the graph. Obviously, the number of vertices does not change, since the same distribution was used everywhere; there are many more edges for the three last methods, because these graphs are transitive.

Then the small world characteristics, clustering coefficient (CC) and average path length (APL). The clustering coefficient is relatively low, but lower for the two transitive graphs (2 and 3) than for the non-transitive graph. This might surprise at first sight, because there are many more edges in these graphs, which should translate in more edges and thus a higher clustering coefficient. However, in a transitive graph, vertices have many more direct neighbours than in a non-transitive graph, so this balances out.

As for the APL, this really is an effective measure only in the first graph; for the other graphs, since they are transitive, path lengths are either 0 (not connected) or 1 (connected). Hence the values lower than 1 in this column.

Then, the components—in this case, when calculating the components edge direction is not considered: their number (Comp), their average size (CpAvg) and the size of the largest component (LComp). Whatever the method used, the distribution is made up of one enormous component and many smaller ones. Using the first method, however, there are far fewer components than using the other two: the reason of this is that using the first method, conflicts are represented by edges in the graph, which means that clusters that are connected by a conflict edge are grouped together.

8.2.1 Hub nodes in the strong dependency graph

We have noted earlier that distributions are *directed* graphs. Most graphs that the small-world principle has been applied to (such as social networks, or the graph of the Web), are *undirected* graphs.

This requires us to modify the notion of a hub. In an undirected graph, a hub is a node with many connections, no matter what the nature of these

connections actually is.

However, in a *directed* graph, the connections to a node can either be incoming or outgoing. A node with many connections can thus fall into one of three categories:

- A node with many *incoming* connections, but few outgoing ones. In a distribution context, a good example of this would be a library.

- A node with many *outgoing* connections, but few incoming ones. In a distribution context, this would be a meta-package—a package that does not contain any files, but only serves to easily install a collection of many packages, such as KDE or Gnome.

- A node with both many incoming connections and many outgoing connections.

Debian

To assess which of these package types actually occur in F/OSS distributions, we have plotted the incoming and outgoing degrees in the strong dependency graph (the normal, transitive version) of Debian-`stable` in figure 8.2.
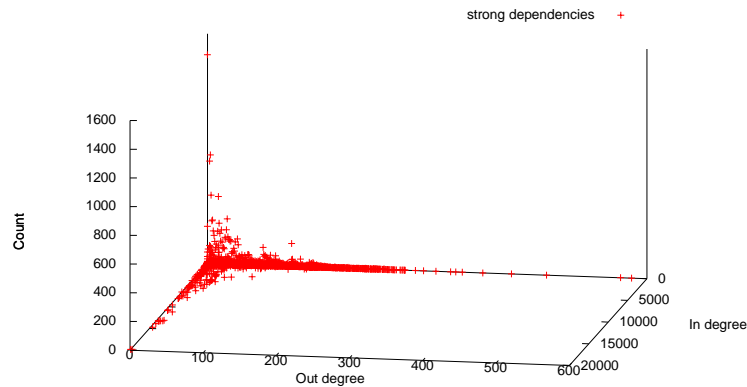
Figure 8.2: In and out degrees in Debian `stable`, strong dependencies

This figure shows clearly that there are a few library packages (high in degree, low out degree), a few meta-packages (high out degree, low in degree), but no packages that combine a high in degree with a high out degree. Schematically, the distribution looks like shown in figure 8.3.

Let us look at the ten packages with the highest in and out degrees (in the strong dependency graph) respectively.
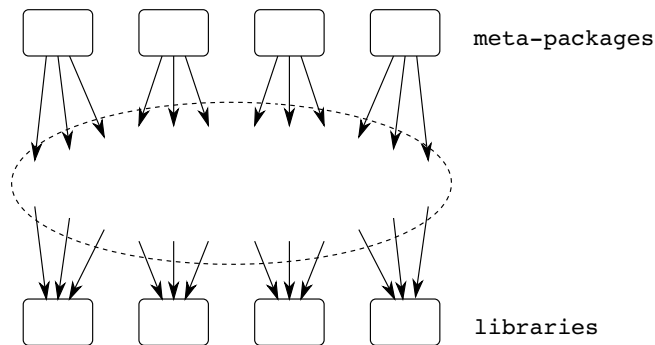
Figure 8.3: Schematic repository structure

Highest in degrees			Highest out degrees		
Name	In degree	Out degree	Name	In degree	Out degree
gcc-4.3-base	19 821	0	kde	0	579
libgcc1	19 819	2	gnome	0	564
libc6	19 819	2	gnome-desktop-environment	1	463
libstdc++6	14 710	3	gnome-devel	0	415
libselinux1	13 884	3	gnome-core-devel	1	376
lzma	13 299	4	gnome-dbg	0	348
libattr1	13 254	3	kde-devel	0	339
libacl1	13 232	4	gnome-office	0	332
coreutils	13 219	6	audacious-plugins-dev	0	312
dpkg	13 215	9	gnome-accessibility	0	295

The first thing we can see is that many of the packages seem to come from the same software suite: the gcc-4.3-base, libgcc1 and libc6 packages, for example, and all the gnome packages. It is reasonable to assume that these packages depend on each other, and thus that some of their impact sets overlap (as noted in the section on dominance of chapter 3).

When we look at the dominator graph, we can see that libc6, libgcc1 and gcc-4.3-base form a cluster (see figure 8.4a), and that dpkg, coreutils, lzma, libacl1 and libattr1 are in the same cluster, together with perl-base (see figure 8.4b).

(a) libc6

(b) coreutils
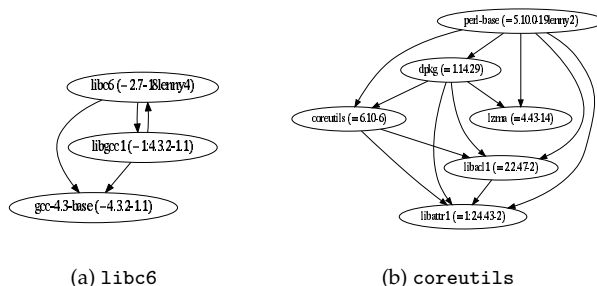
Figure 8.4: Clusters in the top 10 of in degrees in Debian 5.0.6

For the packages with high out degrees, gnome is in a cluster with gnome-desktop-environment, and gnome-devel with gnome-core-devel.

This allows us to 'purify' the top 10 list, by only mentioning one package per cluster (the one with the highest degree). In this way, the table will contain only independent packages, and not packages whose degree is artificially augmented by a dependency on a package that itself has a high degree. Here is the new list:

Highest in degrees			Highest out degrees		
Name	In degree	Out degree	Name	In degree	Out degree
gcc-4.3-base	19 821	0	kde	0	579
libstdc++6	14 710	3	gnome	0	564
libselinux1	13 885	3	gnome-devel	0	415
lzma	13 299	4	gnome-dbg	0	348
debconf	11 144	11	kde-devel	0	339
zlib1g	10 866	3	gnome-office	0	332
libncurses5	10 680	3	audacious-plugins-dev	0	312
libdb4.6	9 581	3	gnome-accessibility	0	295
debianutils	8 097	3	planner-dev	0	284
libgdbm3	8 094	3	koffice-dev	0	284

Just by looking at the package names, our classification seems to be justified: the packages with high in degree seem to be libraries, and the packages with high out degree seem to be high-level metapackages.

There is a way to check this more quantitatively. Most Debian packages have a list of *tags* that indicate the type of package. Let's look at the tags that occur more than once in the packages mentioned above[2]:

High in degrees		High out degrees	
Tag	Count	Tag	Count
role::shared-lib	5	suite::gnome	6
interface::commandline	4	interface::x11	5
devel::library	3	role::metapackage	4
implemented-in::c	3	uitoolkit::gtk	4
role::program	3	special::meta	3
scope::utility	3	x11::application	3
suite::gnu	3	devel::library	2
admin::configuring	2	role::devel-lib	2
implemented-in::perl	2	role::dummy	2
interface::text-mode	2	use::editing	2
suite::debian	2		
use::compressing	2		

We can see that packages with a high in degree are often (shared) libraries, or administration utilities. Packages with a high out degree are often metapackages or dummies, or high-level applications. Our classification is therefore confirmed by Debian's own tagging system.

There is only one tag that appears both right and left: `devel::library`. One should expect this tag for packages with a high in degree, but its presence in packages with a high out degrees is less obvious. The two packages from our top 10 of highest out degrees that have the tag are `planner-dev` and `koffice-dev`.

In fact, the `planner-dev` and `koffice-dev` packages are *development* packages (they both have the `devel::library` tag, but also the `role::devel-lib` tag). They have all the dependencies the normal `koffice` and `planner` packages have, hence the large out degree, but there are no packages that depend on them (packages rarely need the development libraries to function, only the main package itself).

---

[2]Three packages, `libdb4.6`, `kde` and `kde-devel`, did not have any tags at all.

Mandriva

The values mentioned in the previous paragraph have been computed on Debian distributions; in this subsection, we shall try to identify if they are also pertinent for the Mandriva distribution.

This would seem to be the case: the fact that the `libc6` package is very important, for example, is inherent in Linux and should not depend on distribution policies.

Let us see if the figures corroborate this intuition. First the table of measures that are important for small world characteristics.

Method	V	E	CC	APL	Comp	CpAvg	LComp
1	7 566	84 855	0.47	7.49	289	26.18	7 273
2	7 566	1 170 721	0.25	0.94	333	22.72	7 230
3	7 566	721 162	0.25	0.94	339	22.32	7 223

distribution used: mandriva/x86_64 2010.1 main

The meaning of the values in this table have already been discussed in the section on Debian, but we shall recall them quickly: V and E denote the number of vertices and edges in the graph, CC its clustering coefficient, APL the average path length, Comp the number of components (when not taking edge direction into account), CpAvg the average component size and LComp the size of the largest component.

The three methods used for generating the graphs are, respectively: taking any relationship between two packages to add an edge to the graph; taking the fact that there is a dependency path between two packages to add an edge to the graph; and taking the fact that two packages have a strong dependency to add an edge to the graph (see the start of the section for more details).

From the values, we can see that Mandriva has small world characteristics, but the clustering coefficient is lower than for Debian.

This is corroborated by examining the degree distribution plot shown in figure 8.5 (this is the same graph as in figure 8.1, but this time for Mandriva): it is more difficult to see a power law distribution here, because there is much more deviation in the values.

Figure 8.6 contains a plot of the in and out degrees together (it is the same figure as 8.2). This figure shows a clearer image: the distribution of the in and out degrees looks comparable to the distribution in Debian, except that there are several outlying packages with a high out degree.

This is due to two packaging particularities in Mandriva: first, there are several *task* packages in Mandriva that serve to be able to install one set of packages that fulfil a specific function—the X window system, for example—in one go. In Debian, these tasks exist as well, but they are not implemented as packages, but rather as fields in the metadata of existing packages. These task packages are like meta packages in that they have a high out degree, but a small in degree, and this explains the outliers in the graph.

Second, there are simply much more dependencies in Mandriva. A package like `knetwalk`, which is a standard KDE application, has 368 strong dependencies in Mandriva, whereas it has 116 strong dependencies in Debian.

When we look at the top 10 of high-degree packages for Mandriva (it has already been corrected for clustering; see the previous section on Debian for more details on this procedure), we can see many recognisable packages: one
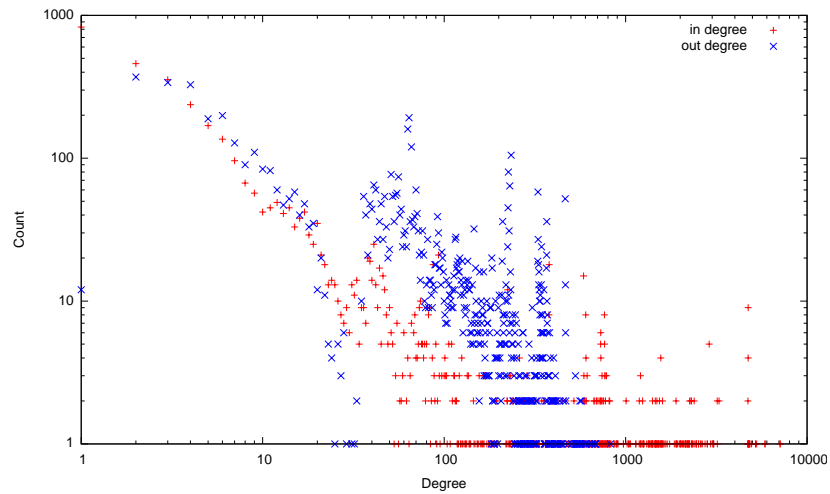
Figure 8.5: Distribution of degrees in Mandriva 2010.1 (method 3)

noticeable absent in the high in degree column, however, is `libc`. This is because the `glibc` package dominates `dash-static` and has therefore been removed (it has an in degree of 7 105).

The presence of the `dash-static` package itself might also seem surprising; however, this is explained by the fact that many Mandriva packages need a shell to run their pre- and post-install scripts; this requirement translates into a dependency on `dash-static` (dash is a very light-weight shell).

Highest in degrees			Highest out degrees		
Name	In degree	Out degree	Name	In degree	Out degree
`dash-static`	7 106	0	`task-kde4-devel`	0	824
`lib64termcap2`	5 862	2	`kimono-devel`	0	683
`perl-base`	5 274	2	`smoke4-devel`	4	675
`libgcc1`	5 206	2	`kdenetwork4-devel`	0	655
`lib64pcre0`	4 836	4	`kipi-plugins-devel`	0	651
`uClibc`	4 719	5	`kdepim4-devel`	0	645
`ncurses`	3 272	5	`kdeplasma-addons-devel`	0	630
`lib64gdbm3`	3 202	2	`nepomuk-scribo-devel`	0	606
`openssl-engines`	3 196	3	`kdewebdev4-devel`	0	605
`lib64xau6`	3 056	2	`kdepimlibs4-devel`	11	599

distribution used: mandriva/x86_64 2010.1 main

Evolution over time

Does the small-world nature of distributions evolve over time? In the following tables, we have summarised the evolution of the relevant properties over time for the Debian and Mandriva distributions. The columns have largely the same significance as in previous tables, to wit: V and E the number of vertices and edges, respectively, in the strong dependency graph; E/V the ratio between these (the number of edges per vertex); CC the clustering coefficient; APL the average shortest path length; ZDP the percentage of zero-degree packages in the strong dependency graph, i.e. the number of packages in the distribution that both do not have any strong dependencies and are not strongly

118
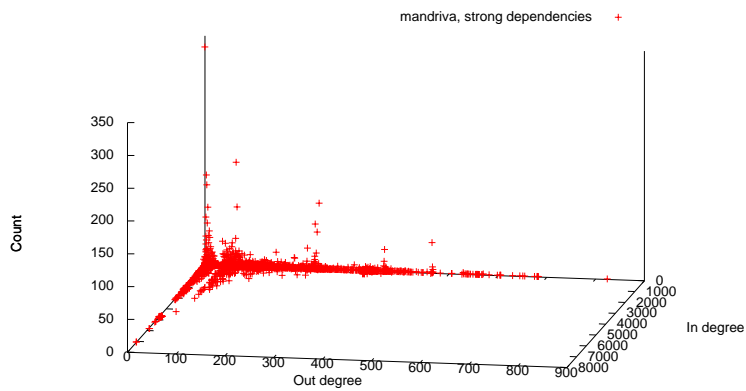
Figure 8.6: In and out degrees in Mandriva 2010.1, strong dependencies

depended on by any package; Comp the number of weakly connected components; CpAvg the average size of the weakly connected components; and LComp the size of the largest weakly connected component.

Distribution	Year	V	E	E/V	CC	APL	ZDP	Comp	CpAvg	LComp
3.0	2002	8 273	87 078	10.52	0.36	0.90	7.81	657	12.59	7 575
3.1	2005	15 195	339 786	22.36	0.34	0.91	6.77	1 056	14.39	15 063
4.0	2007	18 052	510 214	28.26	0.27	0.91	6.75	1 241	14.55	16 744
5.0	2009	22 311	804 486	36.05	0.29	0.91	6.50	1 482	15.05	20 757
5.0.6	2010	22 000	792 977	36.04	0.29	0.91	6.61	1 484	14.82	20 445

distribution used: debian/i386 and debian/amd64 stable

Here are the equivalent numbers for Mandriva:

Distribution	V	E	E/V	CC	APL	ZDP	Comp	CpAvg	LComp
2007.0	5 123	393 514	76.81	0.36	0.97	2.3	121	42.34	5 002
2007.1	5 517	357 386	64.77	0.37	0.96	3.1	175	31.53	5 339
2008.0	6 083	387 377	63.68	0.39	0.95	4.0	248	24.53	5 832
2008.1	6 410	431 378	67.29	0.24	0.94	4.5	293	21.88	6 110
2009.0	6 860	536 402	78.19	0.26	0.94	4.5	311	22.06	6 542
2009.1	6 936	545 064	78.58	0.26	0.94	4.7	330	21.02	6 602
2010.0	7 340	627 801	85.53	0.25	0.94	4.6	344	21.34	6 992
2010.1	7 566	721 162	95.31	0.25	0.94	4.4	339	22.32	7 223

distribution used: mandriva/x86 and mandriva/x86_64 main

There are no great surprises in the tables shown above, except two: the rise in ratio between edges and vertices, and the decrease in clustering coefficient. These two figures are especially interesting in combination; in a random graph, one might expect the two to be correlated: having relatively more edges means that there is a higher chance that neighbours of one vertex are also neighbours of each other, which would result in a higher clustering coefficient.
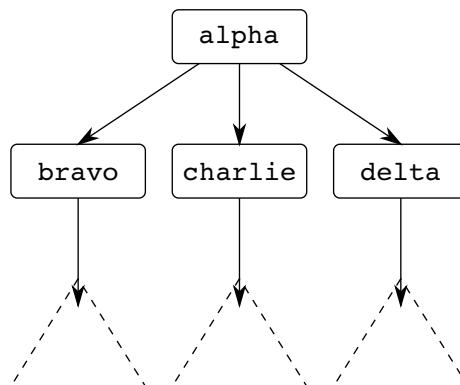
Figure 8.7: Example of the evolution of a repository

The strong dependency graph, however, is not a random graph. The pattern can be explained by the historical evolution of the distributions. The first packages to have been added are generally the most important and oft-used ones: a high proportion of these are libraries.

The packages that are added afterwards are applications that depend on some of these libraries. But because the graph is transitive, any dependency on an intermediate library such as X immediately results in dependencies on every library that is used by X as well—which explains the rise in number of edges relative to the number of vertices.

Furthermore, if the libraries were not already neighbours of each other, they do not become neighbours by such additions; the only edges that are added are the transitive edges between added packages and dependencies—and thus, the clustering coefficient does not increase.

In figure 8.7, we shall show an example that clarifies this. Suppose that packages bravo, charlie and delta are already in the repository, each with their own set of strong dependencies (represented by the dashed triangles). Now, a package alpha is added with strong dependencies on bravo, charlie and delta. Since strong dependencies are transitive, any strong dependencies of bravo (and charlie and delta) are strong dependencies of alpha as well, which considerably increases the number of edges in the graph.

However, the clustering coefficient does not increase; the neighbours of alpha are not neighbours of each other, unless they already were neighbours before the addition of alpha.

The difference between Mandriva and Debian is that the numbers for Mandriva are much more stable. This can be explained by the fact that the time frame for the Mandriva distributions that we have investigated is much smaller; the numbers for Debian change primarily in the first releases, from before 2008.

8.2.2 Hub nodes in the direct dependency graph

The conclusion that there are no packages with both a high in degree and a high out degree holds for the strong dependency graph, which is not the same

as the direct dependency graph. The two cannot be directly compared, though, because the strong dependency graph is transitive, whereas the direct dependency graph is not. Taking the transitive closure of the dependency graph, the graph shown in figure 8.8 is obtained.
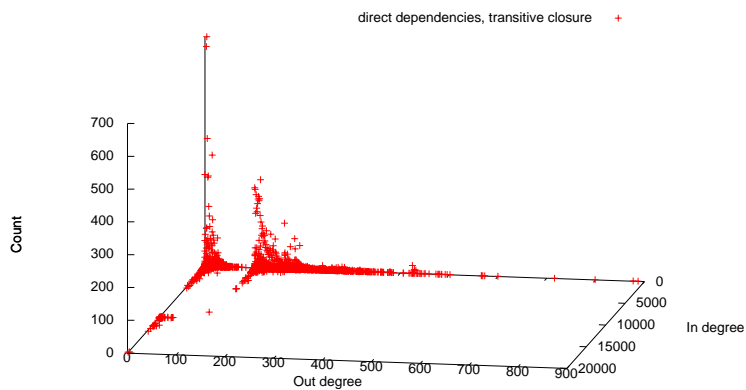
Figure 8.8: In and out degrees in Debian `stable`, direct transitive closure

In figure 8.8, there is an interesting change: the same pattern as for the earlier strong dependency graph is present, but there is a curious echo of this pattern to the right.

By looking at the table, we can see that this 'echo' is generated by packages that are intermediate: libraries that are used often, but that themselves also depend on more primitive libraries such as `libc6`. The first examples found are `xfonts-utils`, `xfonts-encodings` and `x11-common`: all part of the X library, which is used by almost every program that has a graphical user interface, but which itself depends on libraries like `libc6` or `zlib`.

The reason that this 'echo' does not show up in the *strong* dependency graph is that many of the dependency relations mentioned above are disjunctive, and therefore will not be found in the strong dependency graph (`xfonts-utils`, for example, in the transitivised direct dependency graph has an in degree of 11 729 and an out degree of 102, whereas in the strong dependency graph it has an in degree of 121 and an out degree of 23).

8.2.3 Hub nodes in non-transitive graphs

Instead of taking the transitive closure of the direct dependency graph as a basis of comparison with the strong dependency graph, one can also take the transitive reduction of the strong dependency graph and compare it with the original direct dependency graph.

The graph for the (original, non-transitivised) direct dependency graph is shown in figure 8.9. This is the pattern seen earlier in the strong dependency graph: many nodes with a low in and out degree, few nodes with a high in degree, few nodes with a high out degree, and no nodes with both a high in

degree and a high out degree. The only change is in the numbers: the average in and out degrees are lower than before—which is easily explained by the fact that this graph is not transitive and therefore contains fewer edges.
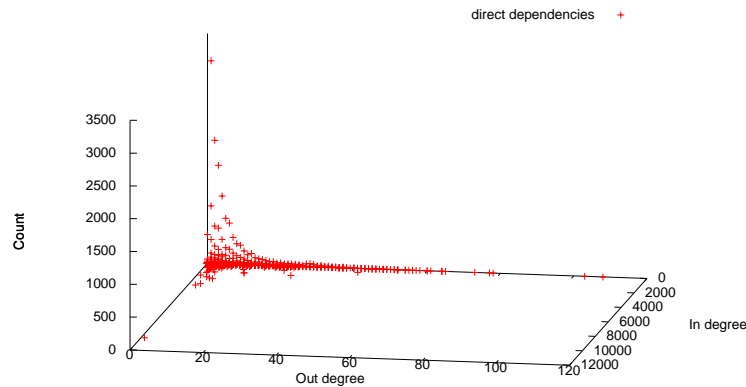
Figure 8.9: In and out degrees in Debian `stable`, direct dependencies

As for the actual tables, the list of packages with a high in degree does not change much with respect to the strong dependency graph: `libc6` is the package with the highest in degree, followed by `libgcc1` and `libstdc++6`.

The list of packages with a high out degree *does* change: the package with the highest out degree when considering only direct dependencies is `gdm`, followed by `texlive-full` and `xserver-org`. This change is caused by the fact that the graph is non-transitive: the `kde` package with has the highest out degree in the strong dependency graph may not have many direct dependencies, but these direct dependencies themselves have other dependencies and so forth.

In order to compare this direct dependency graph with the strong dependency graph, there is still the problem of transitivity. In the previous section we have taken the transitive closure of the direct dependency graph so as to compare two transitive graphs; another option is to take the strong dependency graph and de-transitivise it (i.e. take its transitive reduction). The result of this operation is shown in figure 8.10.

This graph shows much the same pattern as the direct dependency graph (and the transitive strong dependency graph), though here the average in degree has decreased rather a lot: the package with the highest in degree, `libstdc++6`, has 535 incoming edges (in the previous graphs, the highest in degree was in excess of 10 000). Apparently many of the packages that depend on packages like `libc6` also depend on 'intermediate' packages that also depend on `libc6`, thus creating a transitive arc which has now been removed.

There are three rather obvious outliers: three packages that have a relatively high in and out degree. These packages are `libgtk2` (450 in, 11 out), `kdelibs4c2a` (379 in, 17 out) and `qt3-mt` (149 in, 9 out). Like the 'echo' seen in the transitive closure of the direct dependency graph (figure 8.8), these are

122
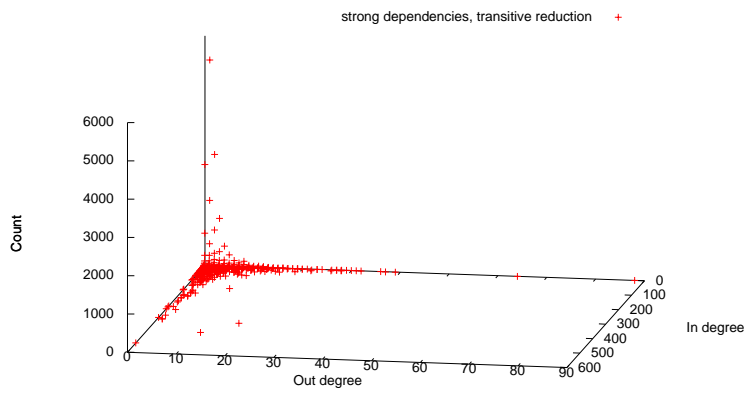
Figure 8.10: In and out degrees in Debian `stable`, strong transitive reduction

intermediate libraries that are used by many packages (Gnome, KDE and Qt respectively) and that depend on many, more primitive, libraries (the X libraries, for example).

Conclusion 9

Now this is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.
— WINSTON CHURCHILL

In this chapter, I will start with a summary of the material presented in the previous chapters, together with some thoughts about its practical usage. Then follows a discussion of related works, together with a discussion of possible future research.

9.1 Summary

The technical matter in this thesis can be divided into four main parts:

1. The definitions of the theory of packages, and the new concepts that are derived from it such as strong dependencies and strong conflicts (chapters 2 and 3);

2. Algorithms that use these definitions, and their implementation (chapters 4 and 5);

3. A formalisation and verification of the definitions (chapter 6);

4. Observations about the general structure and evolution of distributions collected during the application of our tools (chapters 7 and 8).

In the first part, the groundwork has been laid for the rest of this thesis; a mathematical model that allows reasoning about distributions without having to think about the messier details of packaging. Furthermore, the concepts introduced: strong dependencies, dominators and strong conflicts allow us to look at the relations between packages at a higher, more semantic level, which allows us to see errors that would otherwise have been hidden in the enormous amount of data that a distribution contains.

The second part, then, is the application of the first part: how to use the concepts presented to get concrete information about distributions? I have shown how to use certain properties to obtain algorithms that are optimised enough to be run in a few minutes, so that it becomes possible to run them every so often on distributions that can change quickly.

In the third part, some material from the first part is formalised and verified using the Coq proof assistant. The utility of this is fairly obvious: it gives more assurance that the proofs presented are in fact correct.

Finally, the fourth part is more empirical: there is information about the practical run time of the tools, showing that they can indeed be run in a useful timeframe, as well as observations about the constitution of the average F/OSS

distribution. The bulk of these observations centre around the small-world nature of F/OSS distributions: the various characteristics that are connected with this small-world nature, when analysed, provide interesting information about F/OSS distributions.

9.2 Practical usage

The material presented in this thesis is fairly practically oriented, the idea being to adapt known theories in computing science to the specific problem of distribution management. In this section, I will suggest some practical uses for the material presented here.

9.2.1 Theory of packages

In chapters 2 and 3, I have presented a precise, unambiguous specification of the 'theory of packages', developed during the EDOS and MANCOOSI projects. This theory does not depend on any specific package format or distribution, and can therefore be used as a basis for any further works on package management.

Chapter 6 discusses a formalisation in Coq of this theory of packages. This formalisation is not complete (see also the section on future work, later in this chapter), but it could very well be used as a basis for extension efforts.

Another way to use the formalisation is as a basis for formal verification efforts of existing or future package tools.

9.2.2 Tools

The most important practical results from this thesis are the algorithms presented in chapter 4, as well as their implementations, presented in chapter 5.

In part, these tools are already in use with distributions: to name one example, the `distcheck` tool is in use with Debian to make sure that there are no broken packages in its releases.

In section 7.5.3, I have shown another potential usage of the algorithms and tools: daily generation of reports. For the moment, these reports concern only the dominator graph (which primarily serves for analysis of patterns found in the distribution structure), but the reports on strong conflicts could also be useful in this regard: packages with a large exclusion set—which is potentially indicative of errors in the distribution—can thus be identified and eventual errors removed.

9.2.3 Distribution analysis

In chapters 7 and 8, I have presented experimental results obtained by the application of our methods and tools to the Debian, Mandriva and Eclipse distributions.

9.3 Related works

The general field of research on component-based systems is quite extensive, and there is a great amount of literature on the usage of formal methods to improve component interaction. In the following paragraphs, I will discuss some of the literature that explores thematics similar to the ones presented in this thesis.

9.3.1 Predictable assembly

The problem of static analysis of F/OSS distributions is related to the problem of *predictable assembly*, as discussed in [CSSW02]. As defined there, a problem in predictable assembly is a problem that can be reduced to the form: "Given a set of components C, predict property P of an assembly A of these components". Taking C as a distribution, A as a set of packages, and P as the properties described before (installability, strong dependency, etcetera), our problems can be seen as instances of problems in predictable assembly.

In [HMSW01], the concept of *latency* has some connections to our concept of abundance (definition 2.4); the latency of a component depends on the latency of its dependency. Something similar applies in [LWNC02], where a concept of *consistency* is introduced that again resembles our definition of abundance.

9.3.2 Software product lines

A software product line is a family of related programs; the distinction between the different members of a family lies in the features they support.

Features are similar to software packages in that there can be dependencies and conflicts between them: an application might, for example, have support for two mutually exclusive methods of encryption that both depend on a generic cryptographical module.

The standard method to reason about this is by using *feature models* [KCH$^+$90], encoded in feature diagrams [SHT06]. These diagrams are a formal way of specifying the relations between different features, and as such there is a decided similarity between them and the problems discussed in this thesis.

As a matter of fact, it has been shown in [DCZ10] that a significant subset of free feature diagrams can be encoded as a package problem. This would open the way to re-using the results from this thesis in the field of software product lines.

The configuration of a complex piece of software, for example the Linux kernel, can also be seen as a software product line with features, where certain options imply others, or forbid them; this approach is presented in **??**.

Another article that discusses product lines in conjunction with F/OSS distributions is [HRCGB08]. This paper advocates the use of a product line approach by noting that historically, the number of dependencies in Debian distributions grows faster than the number of packages, which will eventually lead to unmanageable distributions.

Though our data (see chapter 7) indicates that in recent distributions, the growth ratio between packages and dependencies has converged, the conclusion does stand that distributions, due to their size, need advanced management techniques.

9.3.3   F/OSS quality assurance

The field of quality assurance of F/OSS distributions has seen a fair amount of growth these last few years.

I should start here by mentioning the EDOS project, and its successor, MANCOOSI, as a part of which project this thesis has been written. The EDOS project (Environment for the development and Distribution of Open Source software), which ran from 2004 to 2007, has studied the problems associated with the production, management and distribution of F/OSS packages. Some of the most important results from this project can be found in this thesis, as well as in [MBDC+06].

One of the results of the EDOS project is the EDOS weather site [pro10], where all current information about non-installable packages in the Debian distribution is condensed into a "weather report", which tells a user the state of the current distribution. This then allows the user to determine whether to run an update, or to postpone it until the distribution is in a better state.

The successor project, MANCOOSI, has continued the work of EDOS, with the results seen in this thesis. The project is larger than this subject alone, however. An example is research done on a complete model for F/OSS package systems, as presented in [DRPPZ09].

Another area of research in the MANCOOSI project is the elaboration of techniques for solvers; in this thesis, we have treated the SAT solver as a black box, but in [LBP08] and [TLO10], the specific problems of solving dependency problems (using SAT solvers and pseudo-boolean optimisation respectively) are treated in more detail.

There has also been work on transactional upgrading and rollback; in this case, if an upgrade goes wrong in some way or turns out not to have been satisfactory, it is possible to go back to an earlier situation. This brings with it a large set of problems, the management of dependencies for example; one article that discusses this is **??**.

Finally, in the project, there has also been some work on optimisation criteria, so that a user can specify in as much detail as possible how to select the packages to install (prefer the newest versions; change as little as possible; etcetera). More information on this can be found in [ALMS09].

The QUALOSS project, another project funded by the European Union, has the objective to create an objective method to assess the quality of open source software. Although this thesis focuses on the quality of open source *distributions*, the subject of QUALOSS complements it quite nicely: after all, the quality of the software that is distributed has an enormous impact on the quality of the distribution itself.

A fourth project funded by the European Union, QaliPSo, has as its motto "Trust and Quality in Open Source Systems". In contrast with the MANCOOSI project, it focuses more on the management side of things; its goal is to facilitate the use of free and open source software by industries and government. This goal is to be reached by defining and implementing the technologies, processes and policies that can be used by all the actors.

This mission has a certain overlap with this thesis, in that the results presented here are technologies that, in providing improved methods for distribution quality assurance and therefore improving the quality of open source distributions, make it easier to use free and open source software.

9.3.4 Distribution structure

As seen in chapter 8, the subject of small-world networks and its application to F/OSS distributions has already been treated in [LW04] and [MSSvK08].

The conclusions drawn in this thesis are similar to the ones drawn in these papers: F/OSS distributions present small-world characteristics.

However, since neither of the papers go into much detail about methodology, it is difficult to draw many conclusions from them—and indeed, the authors do not. They do indicate, however, especially in [LW04] that further study of the subject is necessary.

In [MSSvK08], a very detailed model is created that also takes into account the evolution of the repository.

Here also, no conclusions are drawn about what this actually means for the structure of the repository and its management—the authors merely note that the fact that it is possible to create a model for F/OSS distributions can be used as a first step in creating models for even more complex systems.

Within the broader topic of analysis of the structure of F/OSS distributions, an extensive analysis of the evolution of Debian distribution is presented in [GBRM$^+$09]. In this paper, the authors note that whereas every release is double the size of its predecessor, the average size of packages does not change between distributions, which is consistent with our findings.

Furthermore, the authors present observations about the evolution of Debian distribution, adding data about the usage of different programming languages, the persistence of packages between different releases and the evolution of the package population in terms of lines of code of the source packages.

9.3.5 Other component-based systems

F/OSS distributions are not the only component-based systems where dependencies exist, even though they are amongst the most complex. For example, we have already seen that software product lines also can be analysed using the dependency model.

The Eclipse software development environment been discussed in this thesis; it is similar to F/OSS distributions, though its different scope of application (plugins instead of independent applications) and the difference in quality assurance processes (far less conflicts) make that the challenges are not entirely the same. More detail about the Eclipse environment and its handling of plugins and their relations can be found in [LBR10] and [Boz10].

Another well-known component framework is the OSGi component model [All09], which is used for Java modules instead of software packages. Nevertheless, the ideas are much the same and there is a relatively straightforward translation to SAT as well, as presented in [JDG10].

A different approach to the problem can be found in [VR02]. In this paper, the same general idea is used: a formal component description (in this case, using XML), followed by an automated phase in which dependencies are resolved (using partially ordered multisets). This approach is not oriented towards any specific component-based system, but can be used for any system that involves components and dependencies between them.

9.4 Future work

9.4.1 Tools

At the moment, the `pkglab` tool (see chapter 5) is implemented using `dose2`. An obvious improvement would be to re-implement it using dose3; this re-implementation effort would be a good time to implement some more important changes as well.

First and foremost among these is a redesign of the DQL language, to integrate it more with the underlying API. This would have the effect that if a new algorithm were implemented in dose3, it would be accessible in the DQL without much extra effort.

Another usage would be to revive the `anla` tool, to be an extensive Web-based synthesis of all the results that can be found using the algorithms and methods presented in this thesis. Not only would it be possible to browse distributions on-line, but also to follow their evolution through time.

9.4.2 Algorithm verification

In chapter 6, I have presented a formalisation using Coq of some of the theorems presented in this thesis. This formalisation is not complete; one notable theorem that is still missing is theorem 3.19.

After the formalisation is complete, it can be used to formally verify the algorithms presented in chapter 4, for example using Matthieu Sozeau's PROGRAM extension to Coq.

A major obstacle for such a verification is the fact that one would need a certified SAT solver, which is not the easiest thing to implement.

One way around this using a methodology equivalent to that proposed by Xavier Leroy in [Ler06]. For the certified compiler proposed there, some algorithms are not certified completely; only the results are verified by a certified checker. The idea in our case would be not to provide a certified SAT solver, but only to verify the result of the SAT solver a posteriori.

This approach cannot be translated directly; it is true that a positive result by the SAT solver can be verified quite easily, a *negative* result needs to be verified as well, which can only be done by a SAT check.

However, if our SAT solver gives a negative results, it also gives a reason for this negative result (usually an unsatisfiable dependency or a conflict between two necessary packages). It should be possible to use this reason to verify a negative result.

Another option is the usage of minimal unsatisfiable clauses. A SAT solver using this technique can give a minimal subset of the problem that is unsatisfiable. Since this subset is small, it should be possibly to verify this even with a basic (possibly brute-force) SAT solver, which can easily be certified.

Acknowledgements *A*

List of Figures

List of Algorithms A

Bibliography *B*

Dem, der studiert, um Einsicht *zu erlangen, sind die Bücher und Studien bloß Sprossen der
Leiter, auf der er zum Gipfel der Erkenntnis steigt.*
— ARTHUR SCHOPENHAUER

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(1):47–97, Jan 2002.

[ADCBZ09] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 89–99, Washington, DC, USA, 2009. IEEE Computer Society.

[AJB00] Réka Albert, Hawoong Jeong, and Albert Barabási. Error and attack tolerance of complex networks. *Nature*, 406:378, 2000.

[All09] The OSGi Alliance. OSGi service platform core specification version 4.2, 2009.

[ALMS09] Josep Argelich, Inês Lynce, and Joao Marques-Silva. On solving boolean multilevel optimization problems. In *IJCAI*, pages 393–398, 2009.

[Apa09] Apache Software Foundation. Maven project. `http://maven.apache.org/`, 2009.

[BA99] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[Bai97] Edward C. Bailey. Maximum RPM, taking the Red Hat package manager to the limit. `http://rikers.org/rpmbook/`, 1997.

[BDCV+08] J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, and F. Mancinelli. Improving the quality of gnu/linux distributions. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 1240 –1246, jul. 2008.

[BKM+00] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309 – 320, 2000.

[Boz10] Çagdas Bozman. Converting Eclipse metadata into CUDF. Technical Report TR5.3, MANCOOSI, September 2010.

[CMP00] G. Caldarelli, R. Marchetti, and L. Pietronero. The fractal properties of internet. *EPL (Europhysics Letters)*, 52(4):386, 2000.

[CR08]   Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 3 edition, December 2008.

[CSSW02]   Ivica Crnkovic, Heinz Schmidt, Judy Stafford, and Kurt Wallnau. Anatomy of a research project in predictable assembly. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, 2002. White paper.

[DCB10]   Roberto Di Cosmo and Jaap Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*, pages 163–172, New York, NY, USA, 2010. ACM.

[DCMB+06]   Roberto Di Cosmo, Fabio Mancinelli, Jaap Boender, Jerome Vouillon, Berke Durak, Xavier Leroy, David Pinheiro, Paulo Trezentos, Mario Morgado, Tova Milo, Tal Zur, Rafael Suarez, Marc Lijour, and Ralf Treinen. Report on formal mangement of software dependencies. Technical report, EDOS, 2006.

[DCZ10]   Roberto Di Cosmo and Stefano Zacchiroli. Feature diagrams as package dependencies. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 476–480. Springer Berlin / Heidelberg, 2010.

[DG98]   Debian Group. Debian policy manual. http://www.debian.org/doc/debian-policy/, 1996–1998.

[DML+04]   Scott Dick, Aleksandra Meeks, Mark Last, Horst Bunke, and Abraham Kandel. Data mining in software metrics databases. *Fuzzy Sets and Systems*, 145(1):81–110, 2004.

[DP06]   Debian Project. deb - Debian binary package format. deb(5) manual page, 2006.

[DRPPZ09]   Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Towards maintainer script modernization in foss distributions. In *Proceedings of the 1st international workshop on Open component ecosystems*, IWOCE '09, pages 11–20, New York, NY, USA, 2009. ACM.

[GBRM+09]   Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[HMSW01]   S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau. Packaging predictable assembly with prediction-enabled component technology. Technical Report CMU/SEI-2001-TR-024 ESC-TR-2001-024, Carnegie Mellon University, Software Engineering Institute, 2001.

[HRCGB08] I. Herraiz, G. Robles, R. Capilla, and J.M. Gonzalez-Barahona. Managing libre software distributions under a product line approach. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 1221 –1225, August 2008.

[JDG10] Graham Jenson, Jens Dietrich, and Hans W. Guesgen. An empirical study of the component dependency resolution search space. In *CBSE*, volume 6092 of *LNCS*, pages 182–199. Springer, 2010.

[KAJH99] Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–564, 1999.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU, 1990.

[LBP08] Daniel Le Berre and Anne Parrain. On sat technologies for dependency management and beyond. In *ASPL*, pages 16–19, 2008.

[LBR10] Daniel Le Berre and Pascal Rapicault. Dependency management for the eclipse ecosystem: An update. In *3rd International Workshop on Logic and Search(Lash2010)*, jul 2010.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, 2006.

[Liv05] Benjamin Livshits. Dynamine: Finding common error patterns by mining software revision histories. In *In ESEC/FSE*, pages 296–305. ACM Press, 2005.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[LW04] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[LWNC02] Magnus Larsson, Anders Wall, Christer Norström, and Ivica Crnkovic. Using prediction-enabled technologies for embedded product line a rchitectures. In *Proceedings of CBSE5*, 2002.

[MBDC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208, 2006.

[Mil67] Stanley Milgram. The small world problem. *Psychology Today*, 1(1):60–67, 1967.

[MSSvK08] T. Maillart, D. Sornette, S. Spaeth, and G. von Krogh. Empirical Tests of Zipf's Law Mechanism in Open Source Linux Distribution. *Phys. Rev. Lett.*, 101(21):218701, Nov 2008.

[NNR09] R. Nair, G. Nagarjuna, and A. K. Ray. Semantic structure and finite-size saturation in scale-free dependency networks of free software. *ArXiv e-prints*, January 2009.

[pro10] EDOS project. EDOS weather site. http://edos.debian.net, 2006–2010.

[PvL88] J.A. La Poutré and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *WG '87*, volume 314 of *Lecture Notes in Computer Science*, pages 106–120, 1988.

[Sch08] Michael Schröder. Using SAT for solving package dependencies. In *FOSDEM 2008*, 2008.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *RE'06*, pages 136–145. IEEE, 2006.

[Syr99] Tommi Syrjänen. A rule-based formal model for software configuration. Research Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, second edition*. Addison Wesley Professional, 2002.

[TLO10] Paulo Trezentos, Inês Lynce, and Arlindo L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 427–436, New York, NY, USA, 2010. ACM.

[TZ09] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (cudf) 2.0. Technical Report TR003, Mancoosi, November 2009.

[VR02] Marlon Vieira and Debra Richardson. Analyzing dependencies in large component-based systems. *Automated Software Engineering, International Conference on*, 0:241, 2002.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, June 1998.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, Cambrige, Mass., 1949.