EDOS deliverable WP2-D2.2: Report on Formal Management of Software Dependencies

Roberto Di Cosmo

Deliverable WP2-D2.2

Report on Formal Management of Software Dependencies

Project Acronym	Edos
Project Full Title	Environment for the Development and Distribution of Open Source Software
Project number	FP6-IST-004312
Contact Author	Roberto Di Cosmo, roberto@dicosmo.org
Authors List	WP2 Team
Workpackage number	WP2
Deliverable number	2
Document Type	Report
Version	R.1269
Date	April 21, 2006
Distribution	Public

Authors list

Jaap BOENDER	Université Paris VII
Roberto DI COSMO	Université Paris VII
Berke DURAK	INRIA
Xavier LEROY	INRIA
Marc LIJOUR	Edge-IT
Fabio MANCINELLI	Université Paris VII
Tova MILO	Tel-Aviv University
Mario MORGADO	Caixa Mágica
David PINHEIRO	Caixa Mágica
Rafael SUAREZ	Mandriva
Ralf TREINEN	LSV Cachan
Paulo TREZENTOS	Caixa Mágica
Jérôme VOUILLON	Université Paris VII
Tal ZUR	Tel-Aviv University

Contents

Chapter 1

Executive summary

The stated goal of EDOS Workpackage 2 is

> "to build new generation tools for managing large sets of software packages, like those found in free software distributions, using formal methods"

Our focus is mainly on the issues related to dependency management for large sets of software packages, with a particular attention to what must be done to maintain consistency of a software distribution on the repository side, as opposed to maintaining a set of packages on a client machine.

This choice is justified by the fact that maintaining the consistency of a distribution of software packages is essential to make sure the current distributions will scale up, yet it is also an invisible task, as the smooth working it will ensure on the end user side will tend to be considered as normal and obvious as the smooth working of routing on the Internet.

In other words, we are tackling an essential infrastructure problem, which is perfectly suited for an European Community funded action.

Progress report

Over the first year and a half of its existence, the WorkPackage 2 team of the EDOS project has done an extensive analysis of the whole set of problems that are in its focus, ranging from upstream tracking, to thinning, rebuilding, and dependency managements for F/OSS distributions.

Task 1 (upstream tracking)

The Tel Aviv team has made an extensive study of the event notification systems that can be used to keep package maintainers up to date with software developers,

and showed how tools corrently developed in WP4 are good candidates to solve this problem efficiently and easily.

Task 2 (dependency management)

We have performed an extensive survey of package formats, package management systems and dependency solvers w.r.t. our focus (dependency management, especially for binary packages), and retained only the most significant ones as far as dependencies go, namely Debian packaging, RedHat-like packaging and Gentoo (mostly source) packaging.

All of the commonly used dependency solvers turned out to be either incomplete (in the sense that they do not always find a solution to an installation problem when it exists) or unacceptably inefficient for our applications (some of these tools may take months of intensive computation to find a solution).

This is not really surprising, as we have already shown in D2.1 that the installability problem is NP-complete, and the existing dependency solvers try not only to verify whether a package is installable, but also to find some sort of optimal solution, starting not from an empty set but from the set of packages actually installed on the system. These problems are indeed more difficult.

As a consequence, we had to set out to develop our own tools and algorithms, that are now available to the community through a set of industrial strength, formal method-based tools.

Task 3 (rebuilding from scratch)

We have conducted a careful examination of the build process currently used in Caixa Magica, ex-Conectiva, Debian and Mandriva and we looked at the automated tools and their build system in general.

From this examination, we identified the best practices and the major issues to be addressed have emerged. One is that the compilation environment (e.g. which packages are considered already available at compilation time) can impact the functionalities of the final binary package (e.g. which optional features of the software are enabled by auto-configuration). The second is that the order in which packages are recompiled is crucial and needs to be specified formally and in a way that can be exploited by advanced tools (unlike the shell scripts currently used). The issues with recompilation order are particularly apparent for the software composing the core compilation environment, such as the C compiler and C library, which raise "chicken-and-egg" problems known as *bootstrapping* problems.

We also assessed the tools developed by the EDOS partners, such as Mandriva's `iurt`. The next step will be to finalize these tools and to integrate them in a Linux distributor build system, for example Caixa Mágica and Mandriva's.

Task 4 (thinning)

We performed an analysis of the issues related to building custom distributions, as far as dependency issues go, and we found out that the tools developed in Task 2 seem to be adequate to provide a preliminary approximate solution.

If one fixes more sophisticated goals, like minimizing the size or maximizing the freshness of a custom-built distribution, then it might be necessary to look into more advanced constraint solving and optimization techniques.

Tool development

For the dependency management problems, we performed a detailed review of several solvers for various existing packaging systems, looking at the way they are built and used, and assessing their fitness for use in the framework of the WP2 goals.

This has led to identifying a number of limitations in the current dependency solvers that appear to be either incomplete (in the formal meaning of the term), suboptimal, or inefficient.

For this reason, we have developed a whole set of industrial strength tools, that are nowadays defining the state of the art, as far as single package installability is concerned.

These tools have already been incorporated into the production chain of Caixa Magica, and are in the process of being incorporated in the Mandriva one.

We plan to disseminate widely these tools and make them known to the other communities, like the Debian one.

Formal methods at work

We want to particularly stress here the fact that the tools that have been built and engineered by the team are really based on a theoretically sound formal foundation. In particular, we have:

- formally defined the installability problem and all its related notions, like dependency closure, subrepository etc., using Boolean logic
- formally provided and implemented encodings of the installability problem into a Finite Domain Constraint Problem,
- proved its NP-completeness,

- described and implemented transcodings between the installability problem and SAT (in both directions),

- developed several independent implementations of the verification technique, one fully integrated, in the `debcheck/rpmcheck tool`, and one based on a modularized toolchain, able to call either an Oz-based CP solver, or various SAT solvers (one custom made, and one mainstream, the `fgrasp` solver developed in Portugal).

We did not do a machine-checked proof of correctness of these tools, which is way beyond the scope of the current project, and would require substantial resources. However comparisons between the results produced by our different implementations and the solvers of the package management systems have permitted us to to spot and repair various implementation mistakes and to clarify doubts and ambiguities appearing in the DEB or RPM packaging semantics. As the results of these independent implementation now agree, we are extremely confident in the soundness and completeness of the results provided by our tools.

This is a huge step forward w.r.t. preexisting tools, and we believe to have contributed to a significant advance in the state of the art.

Chapter 2

Overview

Managing large software systems has always been a stimulating challenge for the research field in Computer Science known as Software Engineering. Many seminal advances by the founding fathers of Comp. Sci. were prompted by this challenge (see the book "Software Pioneers", edited by M. Broy and E. Denert [11], for an overview). Concepts such as structured programming, abstract data types, modularization, object orientation, design patterns or modeling languages (unified or not) [26, 18], were all introduced with the clear objective of simplifying the task not only of the programmer, but of the software engineer as well.

Nevertheless, in the recent years, two related phenomena, the explosion of Internet connectivity and the mainstream adoption of free and open source software (FOSS), have deeply changed the scenarii that today's software engineers face. The traditional organized and safe world where software is developed from specifications in a fully centralized way is no longer the only game in town. We see more and more complex software systems that are assembled from loosely coupled sources developed by programming teams not belonging to any single company, cooperating only through fast Internet connections. The availability of code distributed under FOSS licences makes it possible to reuse such code without formal agreements among companies, and without any form of central authority that coordinates this burgeoning activity.

This has led to the appearance of the so-called *distribution editors*, who try to offer some kind of reference viewpoint over the breathtaking variety of FOSS software available today: they take care of packaging, integrating and distributing tens of thousands of software packages, very few being developed in-house and almost all coming from independent developers. We believe that the role of distribution editors is deeply novel: no comparable task can be found in the traditional software development and distribution model. While consulting companies have always had the need to follow actively the development of competing software solutions, we do believe that nothing similar to what a distribution editor does has ever happened before.

This unique position of a FOSS distribution editor means that many of the

standard, often unstated assumptions made for other complex software systems no longer hold: there is no common programming language, no common object model, no common component model, no central authority, neither technical nor commercial[1].

Consequently, most FOSS distributions today simply rely on the general notion of software *package*, which is not to be mistaken for the software organizational unit present in many modern programming languages. A software package is a bundle of files containing data, programs, and configuration information, with some metadata attached. Most of the metadata information deals with *dependencies*: the relationships with other packages that may be needed in order to run or install a given package, or that conflict with its presence on the system.

We now give a general description of a typical FOSS process. In figure 2.1 we have an imaginary project, called `foo`, handled by two developers, Alice Torvalds and Bob Dupont, who use a common CVS or Subversion repository and associated facilities such as mailing lists at a typical FOSS development site such as Sourceforge. Open source software is indeed developed as *projects*, which may group one or more developers. Projects can be characterized by a common goal and the use of a common infrastructure, such as a common version control repository, bug tracking system, or mailing lists. For instance, the Firefox browser, the Linux kernel, the KDE and Gnome desktop environments or the GNU C compiler are amongst the largest FOSS projects and have their own infrastructures. Of course, even small bits of software like `sysstat` consitute projects, even if they are developed by only one author without the use of a version control system. A given project may lead to one or more *products*. For instance, the KDE project leads to many products, from the `konqueror` browser to the desktop environment itself. Each FOSS product may then be included in a distribution. In our example, the project `foo` delivers the products `gfoo`, `kfoo` and `foo-utils`. A *port* is the inclusion of a product into a distribution by one or more *maintainers* of that distribution. The maintainers must:

- Import and regularly track the source code for the project into the distribution's own version control or storage system (this is depicted in figure 2.1 by a switch controlling the flow of information from the upstream to the version control system of the distribution).
- Ensure that the dependencies of the product are already included in the distribution.
- Write or include patches to adapt the program to the distribution.
- Write installation, upgrading, configuration and removal scripts.
- Write metadata and control files.

[1] In the world of Windows-based personal computing, for example, the company controlling Windows can actually impose to the independent software vendors the usage of its own API as well as other rules.

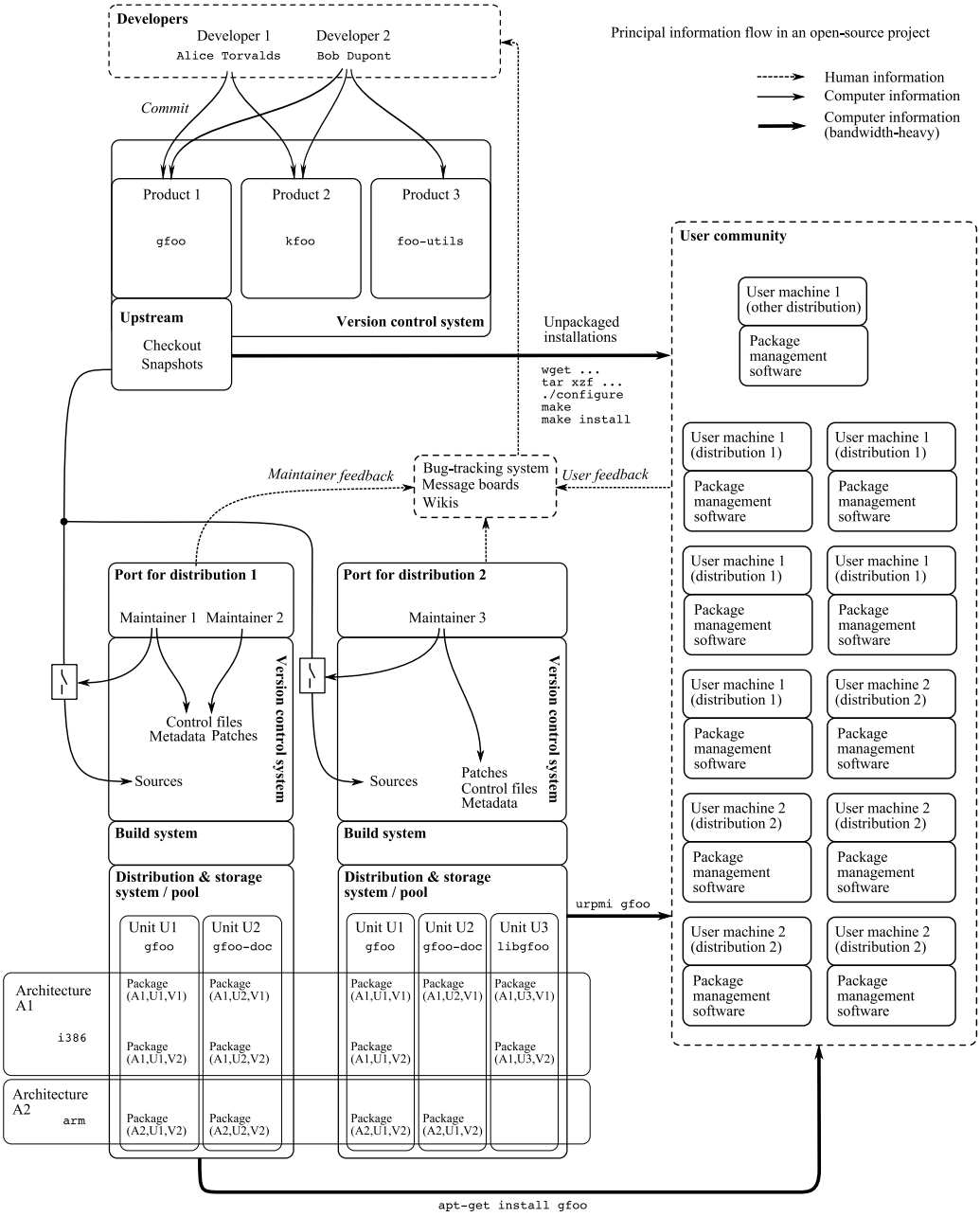


Figure 2.1: Major flow of information in a FOSS project.

- Communicate with the upstream developers by forwarding them bug reports, patches or feature requests.

We see that the job of maintainers is substantial. Attempts to automate some of those tasks, for example by using dependency extraction tools [30, 27] or by getting source code updates from developers [17] are no substitute. In our example, we have a Debian-based distribution 1, with two maintainers for `foo`, and an RPM-based distribution 2 with one maintainer. A given product will be divided into one or more *units*, which will be compiled for the different *architectures* supported by the distribution (a given unit may not be available on all architectures) and bundled as *packages*. The metadata and control files specify how the product is divided into units, how each unit is to be compiled and packaged and on which architectures, as well as the dependency information, the textual description of the units, their importance, and classification tags. These packages are then automatically downloaded (as well as their dependencies) by the package management software (for instance, `apt` or `urpmi`) of the users of that distribution. Some users may prefer to download directly the sources from the developers, in which case they will typically execute a sequence of commands such as `./configure && make && make install` to compile and install that software. However, they then lose the many benefits of a package management system, such as tracking of the files installed by the package, automated installation of the dependencies, local modifications and installation scripts.

2.1 WorkPackage 2's activities in the big picture

We now turn to the problem of ensuring the quality of a distribution, to highlight the focus of WorkPackage 2's activities of the EDOS project.

This problem can be divided into three main subproblems:

Upstream tracking makes sure that the packages in the distribution closely follows the evolution of the software's development. It is almost always carried over by some team outside the control of the distributor (WP2 task1).

Testing and integration ensures that the programs perform as expected in combination with other packages in the distribution. If not, bug reports need to be forwarded to the upstream developers.

Dependency management makes sure that, in a distribution, packages can be installed and user installations can be upgraded when new versions of packages are produced, while respecting the constraints imposed by the dependency metadata (WP2 task 2). In this area also fall activities like building a custom distribution with only a few packages (thinning, WP2 task 3), and making sure that the binary and source packages distributed do match (rebuilding, WP2 task 4).

In this deliverable, we will mainly focus on the last subproblem: dependency management. This task is surprisingly complex [30, 31], owing to the large number of packages present in a typical distribution and to the complexity and richness of their interdependencies.

More specifically, our focus is on the issues related to dependency management for large sets of software packages, with a particular attention to what must be done to maintain consistency of a software distribution *on the repository side*, as opposed to maintaining a set of packages installed *on a client machine*.

This choice is justified by the following observation: maintaining the consistency of a distribution of software packages is *fundamental* for ensuring the quality and the scalability of current and future distributions; yet, it is also an *invisible* task, since the smooth working it ensures on the end user side tends to be considered as normal and obvious as the smooth working of packet routing on the Internet. In other words, we are tackling an essential *infrastructure* problem that has long been ignored: while there are a wealth of client-side tools to maintain an user installation (`apt`, `urpmi`, `smart` and many others [24, 20, 22]), there is surprisingly little literature and publically available tools that address distribution-side requirements. We found very little significant prior work in this area, despite it being critical to the success of FOSS in the long term.

2.2 Task overview

We give now a short summary overview of the main tasks that compose WP2's activity, with a report on their current status.

2.2.1 Task 1: upstream tracking

We believe that the main barrier to scalability in current approaches to upstream tracking is due to the variety and diversity of methods currently used to communicate between developers and package maintainers. To find out about package updates one needs to do one or more of the following: subscribe to relevant mailing lists, follow web pages for announcements, look into CVS/SVN repositories, track software forges, write to the software author, and the like.

Nevertheless, tracking upstream can be cleanly formulated as a simple publish/subscribe problem (as indeed suggested by projects such as Lula), that may receive an adequate solution using the tools developed for WorkPackage 4. The only changes are in the kind and size of user clusters. For this reason, our contribution to Task 1 is via a specific instantiation of WP4 tools, and advances in parallel to the development of these tool in WP4.

2.2.2 Task 2: dependency management

This task turned out to be the most algorithmically challenging: we already proved that a basic subproblem, the test for installability of a single package, is NP-

complete, so our main effort has been on studying actual examples of distributions (Mandriva, Debian and CaixaMagica), and finding efficient heuristics to obtain usable tools.

As a result of this work, we have developed a wealth of tools, whose efficiency is extremely satisfactory: in testing packagewise installability, they outperform any preexisting tool we are aware of, and most of these tools are now either fully integrated, or in the process of being integrated in the production process of CaixaMagica and Mandriva.

The next challenge, much more demanding, is to design tools able to track installability not of a single package, but of full package sets.

During the last semester of 2005, the arrival of Caixa Magica in the consortium also brought to the attention of WP2 the issues related to downgrading (or rollback) of a distribution, a typical *client-side* problem, that seems actually algorithmically related to tracking installability of package sets, a *server-side* problem at the center of our investigation.

2.2.3 Task 3: automatic rebuilding

Most distributions install their software from binary packages compiled by the distribution editor. Building these binary packages from source files can be delicate. Task 3 focuses on streamlining and automating this process, and enabling advanced end-users (and not just the distribution editor) to perform their own rebuilding. The latter can be required for use in high-security contexts, where users do not trust binaries provided by the distribution editor and wish to rebuild their own from sources that have passed a security review.

We have conducted a careful examination of the build process currently used in Caixa Magica, ex-Conectiva, Debian and Mandriva and we looked at the automated tools and their build system in general. From this examination, we identified the best practices and the major issues to be addressed have emerged. One is that the compilation environment (e.g. which packages are considered already available at compilation time) can impact the functionalities of the final binary package (e.g. which optional features of the software are enabled by auto-configuration). The second is that the order in which packages are recompiled is crucial and needs to be specified formally and in a way that can be exploited by advanced tools (unlike the shell scripts currently used). The issues with recompilation order are particularly apparent for the software composing the core compilation environment, such as the C compiler and C library, which raise “chicken-and-egg” problems known as *bootstrapping* problems.

Finally, we re-assessed the tools developed by the EDOS partners, such as Mandriva’s `iurt` and our own WP2 toolkit presented in this document. The next

step will be to finalize these tools and to integrate them in a Linux distributor build system, for example Caixa Mágica and Mandriva's.

2.2.4 Task 4: thinning

Thinning, i.e., the process of extracting a subset of packages from a distribution to build a smaller, specialised custom distribution, is one of the popular businesses in the FOSS world today.

There are some tricky issues here: indeed, the dependency information available in the package metadata is designed to handle installation of packages on an already working distribution, not to install a distribution from scratch.

Hence, extracting a viable subdistribution is not just a matter of getting a conflict-free, abundant component of the dependency graph. One usually resorts to hints like the `essential` metadata field, the `required` priority level and the `base` specification in the `section` metadata field that somewhat fuzzily define the *"base-system"* category in Debian based distributions.

One can reuse the installability algorithms developed in task 2, to get an approximate solution to the problem, but if we want to fix goals like minimum disk size, most up-to-date packages, or the like, some optimization techniques will need to be deployed, and then new algorithmic challenges may arise, that will be similar to the ones already faced, and not really solved, by client-side package management meta-tools.

The following chapters present in detail the analysis and study of each of the problems related to the tasks described above, with a particular focus on Task 2, for which the companion deliverable 2.3 contains a wealth of industrial strength tools developed by the WP2 team over the first half of the project time.

Chapter 3

Upstream tracking

3.1 WP2 - Task 1

3.1.1 Introduction

The goal of Task1 is to keep Linux distribution packages up-to-date with recent source changes. This requires the design of a framework that enables Linux distribution editors (or other interested end users) to be notified automatically of the release of new/updated software packages. Since 2001 there has been a significant amount of research in the area of event-notification systems, leading to the development of several implementations which can in principle serve the EDOS needs. However, analogous to what we have seen already in WP4, the specific EDOS context provides particular environment conditions where better solutions can be developed.

Specifically, we have observed that the packages (that act here as the *topics on interest* in the event-notification framework) naturally divide into *clusters* which correspond to user interests. We have examined the possibility of exploiting such a clustering for improving the performance of the event-notification mechanism, and concluded that it can indeed reduce significantly the network traffic cost. Based on this idea, we are developing *Tamara*, a novel clustering-based event-notification system, designed for topic-based publish/subscribe applications. Tamara can automatically adjust itself to the changes in users interests and provides a self-tuning P2P efficient solution to the package-updates notification problem in the context of EDOS.

The remainder of this section is organized as follows. We start by a brief description of the main principles of existing event-notification systems, focusing on topic-based systems. Then, we consider the particular EDOS setting which led to the design of our novel clustering-based notification system Tamara. We describe it and the research conducted thus far, and conclude with a summery of our ongoing work.

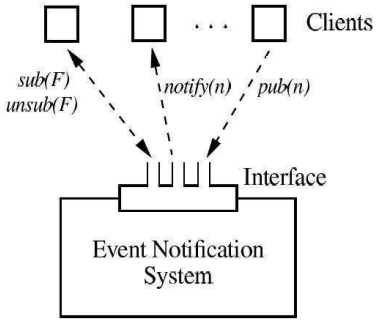


Figure 3.1: Packages clustering.

3.1.2 Existing solutions

An event-notification system is essentially a publish/subscribe system where subscribers are notified whenever data that matches their interests is published. These systems are often classified as one of the following two types: (1) Topic-based systems (e.g. Scribe[13]) where users subscribe to predefined topics of interest, and (2) Content-based systems(e.g. Gryphon[7], Siena[12]), where users declare their interest via a query, and get notified of the publication of new data items whose content satisfy the query predicates. In the context of EDOS, packages naturally play the role of topics of interest. Since topic-based systems are typically easier to manage we have decided to focus our attention here on topic-based systems.

As mentioned above, current topic-based publish/subscribe systems allow users to declare their interest in a topic (or set of topics). The system subsequently notifies the users of any updates or news generated by the topic publisher. Such an update or news is termed an *event*. The act of delivering it to all the subscribed users is termed *notification*.

The common implementation of publish/subscribe systems relies on an *event-notification service* providing (1) storage and management for subscriptions and (2) efficient delivery of events. Such a service represents a neutral mediator between publishers, acting as producers of events, and subscribers, acting as consumers of events. Subscribers register their interest in topics by calling a *subscribe* operation, possibly without knowing the publishers of these topics (see Figure 3.1). This subscription information is recorded in the system and is not forwarded to publishers. To generate an event, a publisher calls a *publish* operation and the system disseminates the event to all subscribed users.

The design principles guiding the existing systems are the following.

- **P2P decentralized solutions.** Scalability is an essential criteria for the rapidly growing Web community. As Server/Client architectures often fail to achieve sufficient scaling, the P2P approach becomes a suitable alternative.

- **Push-based solutions.** Polling of information, even for topics that are frequently updated, is often inefficient as it involves many active queries sent

by participating users, thereby increasing significantly the network traffic. Polling also raises the delicate (and yet not answered) question of *how often should I ask for new updates?* Push-based notification techniques provide a cheaper, more efficient alternative. They minimizes the load on the network and save the need to retransmit the same event again and again to different users.

There are several existing systems with the above characteristics (e.g. Bayeux[25], Gryphon[7], Siena[12] and Scribe[13]). As we will see below, the solution that we propose for EDOS is generic in the sense that it can be built on top of any of these existing systems. As a specific instance we have chosen to use Scribe in our implementation and experimental evaluation. Scribe is a large-scale event-notification infrastructure for topic-based publish-subscribe applications. It supports large numbers of topics, with a potentially large number of subscribers per topic. Scribe is built on top of Pastry, a generic peer-to-peer object location and routing substrate overlay on the Internet, and leverages Pastry's reliability, self-organization and locality properties. Pastry is used in Scribe to create topics and to build an efficient multicast tree for the dissemination of events to the topics subscribers.

We proceed by describing our solution. We first describe the particular properties of the EDOS environment and then explain how our solution fits this particular context.

3.1.3 Clustering-based event-notification system

Our first observation was that in many cases topics of interests are not independent one of another - a user that subscribes to a given topic often also subscribes to related topics. This in particular is the case for EDOS. Dependencies between packages translate here to dependencies between the topics corresponding to the respective packages. Such dependencies have indeed already been exploited in WP4 to cluster packages together and speed up their download time. Our thesis is that they can be also used here to improve the notification of packages updates. To understand this, let us first overview briefly our finding in WP4 and then explain how they can be adapted to the needs of WP2.

WP4. In WP4 the goal is to speed up the delivery of Linux distributions to users. In that context, the open source packages (e.g. Gnome, Gimp, Firefox) play the role of *topics* of interest. The Linux distributions (e.g. Mandriva, Debian, Caixa Magica) and the developers themselves play the role of *publishers*. Finally, the millions of end users are the *subscribers*.

Our research showed that packages can be divided into clusters reflecting the user interests (see Figure 3.2). As explained above this is due to dependencies between packages (e.g. KDE uses QT) and to strong domain coherence (e.g. games, communication utilities, graphics etc). Based on this clustering we devised in WP4

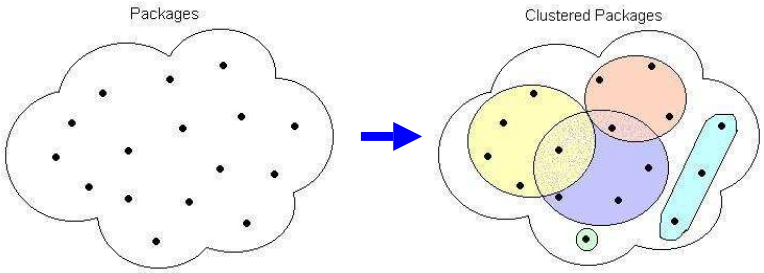


Figure 3.2: Package clustering.

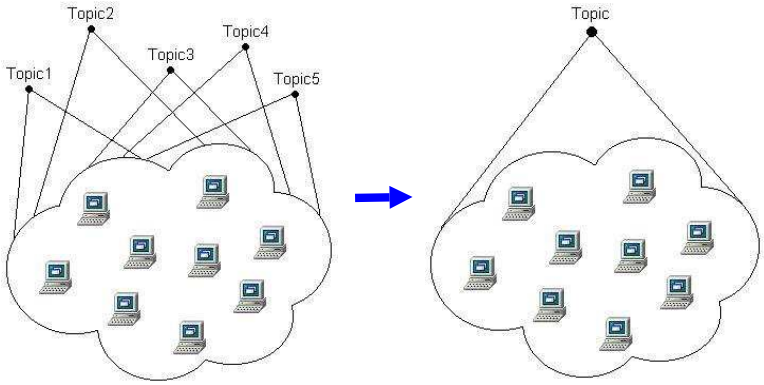


Figure 3.3: Topic clustering.

methods to speed up the delivery of software by grouping packages according to clusters, with the dissemination of each cluster being faster than that of the sum of individual packages.

WP2. For WP2 the picture is similar. Here too the open source packages (e.g. Gnome, Gimp, Firefox) play the role of *topics*. The thousands of Linux developer teams/programmers play here the role of *publishers*. The Linux distribution editors (e.g. Mandriva, Debian, Caixa Magica), as well as any other interested users, play the role of *subscribers*.

Here again the packages divide naturally into clusters that reflect that Linux distributors interests and existing packages dependencies. The two main gains that can be obtained by clustering, in the context of event-notification, are the following.

Less structure maintenance The aggregation of related topics under one common cluster can reduce the maintenance cost of the system's underlying data structures. Instead of managing several structures, one per topic, clustering allows to manage a single structure per cluster (see Figure 3.3). When we have thousands of available topics (as is the case for EDOS), this implies that we only need to maintain a modest number of structures, instead of thousands.

Less overhead for users A common practice in P2P systems is for users to co-operate and "help" each other. In the context of event-notification systems, users (nodes in the P2P network) might be "recruited" to assist in the dissemination of events for topics to which they are not subscribed. When each topic is disseminated independently, users may be recruited to help with the dissemination of any such topics. The aggregation of topics into clusters, reduces the number of dissemination missions and thereby the number of users required.

3.1.4 Dynamic clustering

As explained above, clustering of topics can improve the system performance. However, since we are working in a dynamic environment where users may change their interests, it is important to adjust the clustering dynamically to match the changing environment. The challenge naturally is to do this in a decentralized P2P manner, with maximal system throughput and minimal overhead. This is achieved in *Tamara* using the following two modules.

- **Clusters manager** This module is responsible for the registration of users to clusters and for the dissemination of events, generated by the the topic members of the cluster, to the subscribers.

  Implementation-wise, this module is built on top of an existing standard event-notification system. Each network node holds an internal data structure and uses a system utility, called PMS (Package Mapping System), to get its registration decisions. This is then coupled with a standard topic-based notification system to execute the subscriptions and deliver the events of the topics in the cluster its subscribers.

- **Dynamic clustering manager** This module is responsible for tracking changes in users interests, and adjusting the clustering accordingly. Since we operate here in a distributed P2P environment, with no central complete knowledge of the users interest, we employ a probabilistic model whose main goal is to perform changes only when they have a high of improving the system's state. An important parameter in the decision process is the need to minimize the overhead of the tuning process.

Tamara runs over a standard topic-based event-notification system and DHT. In our implementation, we chose to use Scribe and Pastry[23], respectively, for these functions. Figure 3.4 provides a general picture of the system, when employed in the EDOS context.

Part (a) of the figure depicts the general network structure, including the list of available packages and created clusters whose information is stored in a distributed P2P fashion (via DHT) among the participating nodes. Part (b) depicts the system architecture, where Tamara uses both Scribe and Pastry as operational layers. Part (c) draws a sketch of Tamara's internal data structure and main system.

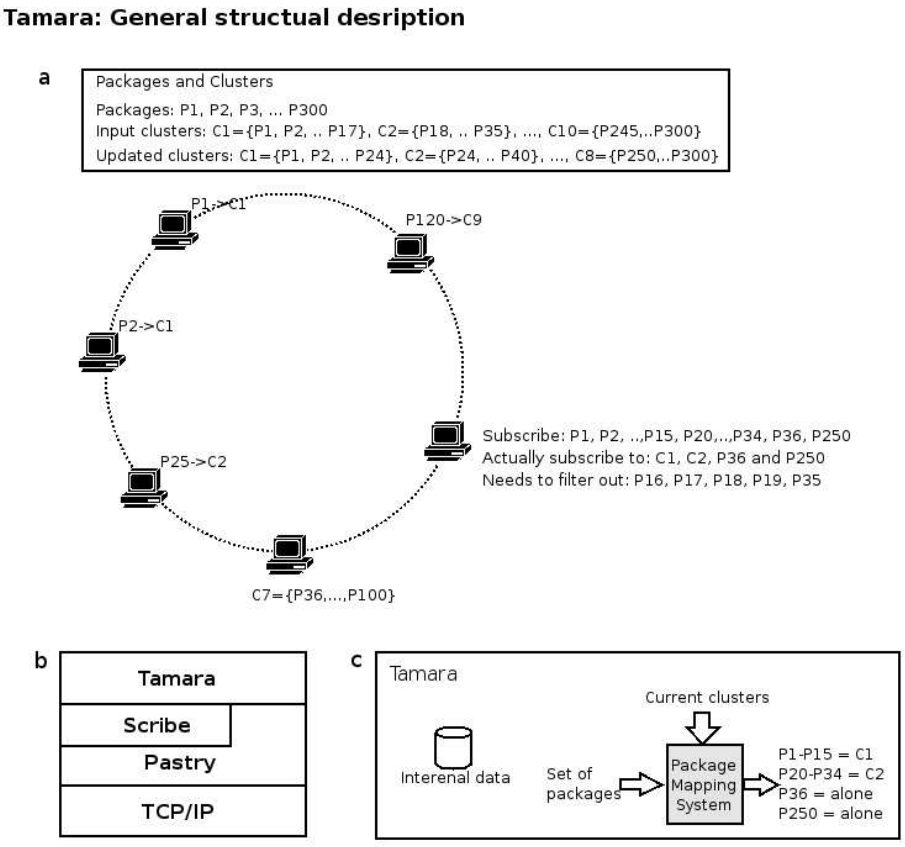


Figure 3.4: System Architecture.

3.1.5 Simulations

Before starting the implemention of *Tamara* we wanted to validate our approach and estimate the potential benefit it can bring in terms of reduction in communication overhead. We have built a system simulator and run a significant amount of experiments on it. In these we measured the three main components which influence the system's performance:

- **Data structure maintenance cost.** The dissemination of events, in each topic/cluster, is supported by an underlying data structure (a multicast tree) that is constantly maintained by the system. The size of the data structure for each topic/cluster, and consequently its maintenance cost, is essentially linear in the number of subscribers. The total maintenance cost is the sum of these individual maintenance costs.

  As we will see, clustering reduces the number of underlying data structures that need to be managed. At the same time, due to the overlap between users interest, the number of subscribers to each cluster is not much larger than the number of subscribers to the original individual topics. Consequently we observe significant savings in structure maintenance overheads.

- **Event dissemination cost.** The dissemination cost of a given event is proportional to its number of receivers. In our experiments, the cost measured for each topic/cluster reflects the frequency of events in this topic, the number of clusters to which this topic belongs to and the number of subscribers to each such cluster.

  Observe that, as clusters are not disjoint, the number of subsribers to which events of a given topic are potentially disseminated can be larger with clustering. We will see however that a good clustering entails that the number of such additional messages is low since the users registered to a given cluster are likely to be interested in most of its topics.

- **Clusters maintenance cost.** This measures the communication overhead due to the maintenance of clusters and the corresponding adjustment to users subscriptions.

We briefly report here a representative sample of the results where the simulation was run over synthetic data with 1500 users, 303 topics, and 8 clusters (generated by the clustering algorithms developed in WP4). We compared the performance of event-notification via standard Scribe to that of Tamara's.

Experiment 1 To separate the cost of the event notifications themselves from the underlying cost of data structures and clustering maintenance, our first experiment simulates thirty days of a *silent* environment, where no events are disseminated. In this experiment, each topic (in Scribe) and cluster (in Tamara) conducts a routine

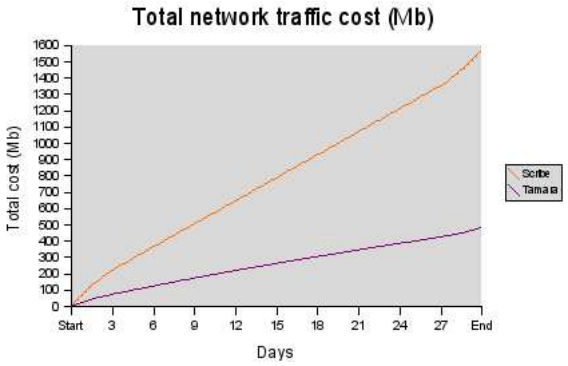


Figure 3.5: Experiment 1(a)

Figure 3.6: Experiment 1(b)

structure maintenance process every three minutes. Every one hour we randomly select one hundred users and change their topic interest.

This experiment aims to confirm our fundamental claim: maintenance of data structure s has a significant effect on the total cost, hence its reduction via clustering significantly reduces the communication overhead. Furthermore, performance is improved by dynamically adjusting the structures to the users changing needs.

In figure 3.5 we can see the total cumulated traffic cost, and in figure 3.6 we see the data structure maintenance overhead, measured every three days.

Experiments 2 and 3 Here the simulations are performed in a similar environment, but every minute we generate and disseminate two random events (experiment 2) or 120 random events (experiment 3). Figures 3.7 and 3.8 depict the results. (We omit here the data structure maintenance cost as in both cases it is similar to that of experiment 1). The total network traffic cost here combines the three cost components detailed above.

Figure 3.9 summarizes the results of the three experiments, showing the performance advantage of Tamara's clustering-based approach.

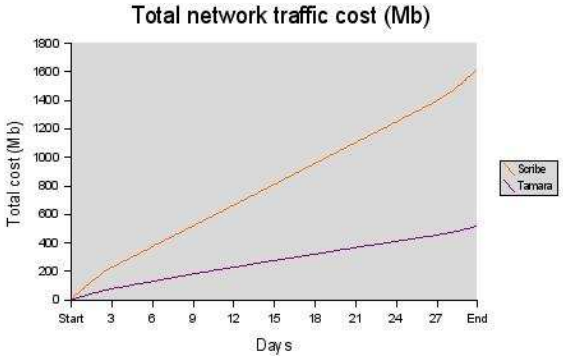


Figure 3.7: Experiment 2

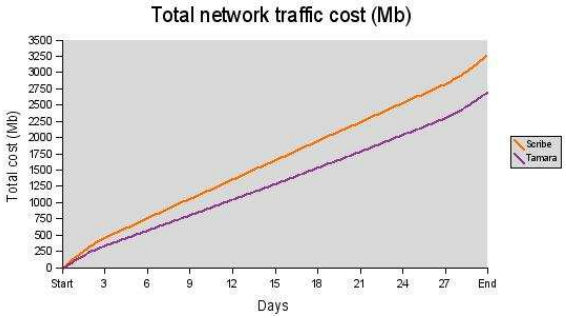


Figure 3.8: Experiment 3

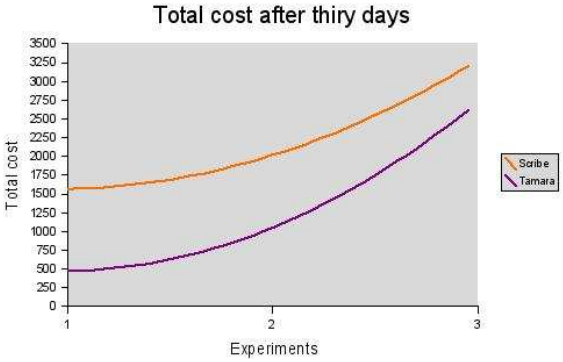


Figure 3.9: Summary of results

3.1.6 Ongoing and future work

Encouraged by the experimental results, our work continues in several complementary directions. First, to further validate our thesis, we continue our simulation experiments, this time using real Linux data (logs and statistics on package updates and their frequencies) obtained from Mandriva. We also test different probabilistic functions (that dictated, in our algorithms, the change policy to the clustering) to optimize the proposed framework.

In parallel, the implementation of Tamara is ongoing. We are currently defining the basic API functions of Tamara and use this specification to implement a first Tamara prototype, running on top of Scribe and Pastry.

Chapter 4

Thinning

Thinning a distribution consists of building a custom distribution for a very specific purpose. Such a distribution is a self-contained set of packages with all the dependencies satisfied, plus the necessary packages to get the system running. Today there is a very extensive market for custom distributions, such as set-top boxes like TiVO, routers like LinkSys, LAMP servers (Linux Apache Mysql and PHP), PDAs like Zaurus, POS, etc., all of which use tailored Linux distributions.

4.1 Dependencies are not enough

The problem is that it is not enough to simply compute the dependencies of the package P of the customer, install them in a system and get it up and running. There are a dozen or so other packages (kernel, binutils, initscripts,...) needed also that are not explicitly included in the dependency closure.

This can be seen by simply executing

```
apt-cache show libc6
```

which gives

```
Depends: libdb1-compat
```

After investigating the result,

```
apt-cache show libdb1-compat
```

one will see that

```
Depends: libc6 (>= 2.2.5-13)
```

So, these two packages, which are the basis for any Linux distribution, depend on each other and nothing else, in particular, not on a kernel or other essential package for the system for that matter.

4.2 State of the art

4.2.1 Linux From Scratch

Linux from Scratch is a set of documents and helper scripts that shows how to create a Linux distribution. The entire system is compiled from scratch and due to this fact it can be tailored to fit any need. This is the basis for any Linux distribution [8].

4.2.2 The debootstrap tool

Debootstrap bootstraps a basic Debian system of any available flavor (eg, sarge, etch, sid) into a target from any debian mirror. Debootstrap is used to install Debian in a system without using an installation disk but can also be used to run a different Debian flavor in a chroot environment. This way a full (minimal) Debian installation can be created, which can be used for testing purposes.

4.2.3 Rpmstrap tool

Rpmstrap is a tool for bootstrapping a basic RPM-based system. It is inspired by debootstrap, and allows you to build chroots and basic systems from RPM sources. At present rpmstrap can build basic Fedora Core 2, Fedora Core 3, Fedora Core 4, Yellowdog 4, CentOS 3, CentOS 4, Mandriva and Scientific Linux systems. It also has support for custom RPM-based systems managed by PDK.

4.2.4 Componentized Linux and PDK

Componentized Linux (CL) is a modular, highly customizable Linux distribution that can be used to build customized versions of the Linux operating system.

Componentized Linux is based on the DCC Common Core, an LSB 3.0 compliant, Debian 3.1 (”sarge”) based common core designed to serve as the basis for custom Debian distributions.

Componentized Linux is accompanied by the Platform Development Kit (PDK), a full suite of tools, best described as “version control for distributions”, for building and maintaining custom distributions using Componentized Linux. The combination is a platform for building custom distributions that provides developers with a set of reusable building blocks, called components, that can be easily assembled into a wide variety of configurations and customized as necessary.

A component is a collection of packages that is internally consistent, along with associated metadata. “Internally consistent” means that all dependencies must be satisfied within the component itself, or the component must explicitly depend on some other component that provides them.

A component has both abstract and concrete representations. The abstract representation is an XML file that specifies the packages in the component along with

metadata such as the component's name. The concrete representation is an APT repository and is automatically generated from the XML component specification.

4.2.5 Catalyst tool

Gentoo is a Linux distribution inspired by the FreeBSD ports system. Almost every package in Gentoo is compiled by the end user before being installed. Keeping this in mind, the Catalyst tool can be viewed as a tool that takes the LFS concept further.

The Catalyst tool is used to build official Gentoo stage tarballs, packages and install CDs. It is also used in other Gentoo projects, such as GNAP and the upcoming Gentoo GameCD project. The goal of the Catalyst project is to provide a single multi-faceted tool that can reliably build all aspects of a Gentoo Linux release: stage tarballs, GRP package sets, and install CDs.

4.2.6 Klik

Klik provides an easy way to download and use software for most major distributions. The approach is to create a self-contained package that is able to run on almost any Linux distribution. Klik is an application that is integrated within the web browser. When a user follows a "klik" link, the software client contacts the server and requests a "klik recipe". The recipe is a file that tells the client what packages to download, the location of these packages and how to install them in a self contained package.

The self contained package is a compressed image file that contains the basic system necessary for the application to run. After it is generated, it is mounted on the loop device and a wrapper script is called that enables the execution of the application.

4.3 Current limitations

The previously mentioned solutions do not easily allow one to build a custom distribution.

LFS allows one to fully tailor a solution for a specific need, but at the expense of manually compiling each and every application from scratch with manual dependency resolution.

Catalyst takes LFS a little bit further with automatic compiling and dependency resolution using "portage" as its package manager, but it allows only the creation of Gentoo releases, GRP package sets and install CDs. It is very specific for Gentoo.

Rpmstrap and debootstrap are two similar tools with the same goal: installing a base system. Although the base system is smaller than a normal Linux installation, the problem is that one has to manually specify which packages not to install from a full list of hundreds of packages to get the system tailored to one's needs.

Rpmstrap allows only the base installation of certain distributions that are preconfigured into the tool. In order to extend the tool for other distributions, one

would have to write specific install scripts for these distributions, for example Caixa Magica.

Componentized Linux and PDK's approach to customizing distributions is quite interesting: they create components, which have a close relationship with packages or sets of packages, and have full dependency closure. But with this solution, one has the burden of maintaining both package metadata and the component's metadata, which in most cases are the same. Although the solution should be a tool which is easier to use and maintain, this tool has many features that should be taken into account.

On the other hand, Klik provides a self contained package with full dependency closure and no external dependencies except a running kernel and possibly a running X server. Although Klik does not allow building a custom distribution, it lays the foundations for tailoring a Linux distribution.

4.4 Proposed approach

4.4.1 The need for additional information

One needs to define a family of "minimal distribution virtual packages", sometimes also called "patterns" in distributions like Suse, or "components" in Componentized Linux. Representative examples of such virtual packages include $B_1 =$ base workstation, $B_2 =$ simple server, $B_3 =$ headless firewall, $B4 =$ router, ... Each virtual package points to the basic packages necessary for the system to be bootable and usable. Moreover, these virtual packages are generic in the sense that they do not specify exact package versions, but only the package names or the required features. For example, they should contain a feature `kernel` , provided by all actual kernel packages, but not `kernel-2.6.15_3` which is too specific and could later conflict with some other chosen package.

4.4.2 A first solution

Having these patterns at hand, building a custom distribution for package P means taking the dependency closure of $P + B_i$ for some B_i, and getting an installation candidate for all packages in this closure. This will mean, for example, that we will have to find a real package `kernel-2.6-something` to satisfy the installability of the virtual package `kernel`.

We now give an example using the `history` tool which integrates our `debcheck` dependency solver. Let `$need` be a set of units that we want to include in our minimal distribution. For instance, take `$need <- { ocaml, vim, ledit }`. The latest available versions of these units can be computed with

```
$latest_need <- latest($need)
```

which gives `{ ledit'1.11-7, ocaml'3.09.1-3, vim'1:6.4-007+1 }`. The set of essential packages can be obtained

```
$essential_packages <- filter(packages, $p -> is_essential($p))
```

but it may contain multiple versions of a given unit. Hence

```
$essential_units <- unit($essential_packages)
```

gives the set of essential units. (At this point, it should be noted that a unit that is essential in a given archive at a given date may not be essential later or in other archives. In order to be rigorous, one should restrict the set of packages to the contents of an archive on a given date.) This set is

```
{ base-files, base-passwd, bash, bsdutils, coreutils,
  debianutils, diff, dpkg, e2fsprogs, findutils, grep, gzip,
  hostname, login, mount, ncurses-base, ncurses-bin, perl-base,
  sed, sysvinit, tar, util-linux }
```

Our set of target units is thus

```
> $target <- $essential_units | $need
```

We can now invoke the dependency solver:

```
$thinned <- install(latest($target), packages)
```

This gives a set of 72 packages:

```
{ base-files'3.1.11, base-passwd'3.5.11, bash'3.1-3,
  binutils'2.16.1cvs20060117-1, bsdutils'1:2.12r-8,
  coreutils'5.94-1, cpp'4:4.0.2-2, cpp-4.0'4.0.2-10,
  debconf'1.4.71, debconf-english'1.4.71, debianutils'2.15.3,
  diff'2.8.1-11, dpkg'1.13.16, e2fslibs'1.38+1.39-WIP-2005.12.31-1,
  e2fsprogs'1.38+1.39-WIP-2005.12.31-1, findutils'4.2.27-1,
  gawk'1:3.1.5-2, gcc'4:4.0.2-2, gcc-4.0'4.0.2-10,
  gcc-4.0-base'4.0.2-10, grep'2.5.1.ds2-4, gzip'1.3.5-12,
  hostname'2.92, initscripts'2.86.ds1-12, ledit'1.11-7,
  libacl1'2.2.35-1, libattr1'2.4.31-1,
  libblkid1'1.38+1.39-WIP-2005.12.31-1, libc6'2.3.6-3,
  libc6-dev'2.3.6-3, libcap1'1:1.10-14,
  libcomerr2'1.38+1.39-WIP-2005.12.31-1, libdb4.3'4.3.29-4.1,
  libgcc1'1:4.0.2-10, libgdbm3'1.8.3-2, libgpmg1'1.19.6-22,
  libncurses5'5.5-1, libncurses5-dev'5.5-1,
  libpam-modules'0.79-3.1, libpam-runtime'0.79-3.1,
  libpam0g'0.79-3.1, libselinux1'1.28-4, libsepol1'1.10-2,
  libslang2'2.0.5-3, libss2'1.38+1.39-WIP-2005.12.31-1,
  libuuid1'1.38+1.39-WIP-2005.12.31-1, libx11-6'6.9.0.dfsg.1-4,
  linux-kernel-headers'2.6.13+0rc3-2, login'1:4.0.14-7,
  lsb-base'3.0-15, mount'2.12r-8, ncurses-base'5.5-1,
  ncurses-bin'5.5-1, ocaml'3.09.1-3, ocaml-base'3.09.1-3,
```

```
ocaml-base-nox'3.09.1-3, ocaml-interp'3.09.1-3,
ocaml-nox'3.09.1-3, perl-base'5.8.8-2, sed'4.1.4-5,
sysv-rc'2.86.ds1-12, sysvinit'2.86.ds1-12, tar'1.15.1-4,
tcl8.4'8.4.12-1, tk8.4'8.4.12-1, util-linux'2.12r-8,
vim'1:6.4-007+1, vim-common'1:6.4-007+1, vim-runtime'1:6.4-007+1,
x11-common'6.9.0.dfsg.1-4, xlibs-data'6.9.0.dfsg.1-4,
zlib1g'1:1.2.3-10 }
```

By adding a bootloader and kernel, we can get a complete, minimal debian system able to run `vim`, `ocaml` and `ledit`.

4.4.3 Further optimizations

Nevertheless, the candidate built as explained above is not optimal, because we want not just a solution, but a solution optimizing metrics such as total size, total number of packages, freshness of versions, software maturity etc. Optimization problems tend to be harder than constraint satisfaction problems, which themselves are harder than purely boolean constraint satisfaction problems. The latter point is illustrated by the problems encountered during our initial approach to solving dependency problems with a general-purpose non-boolean constraint-satisfaction language such as Oz. Severe performance problems can appear if one is not careful when implementing optimization algorithms; see for instance the discussion on Smart in chapter 7.

Chapter 5

Rebuilding from scratch

Linux distributors such as Mandriva or Caixa Magica use their expertise to provide medias containing a self-installable Linux distribution. Like other operating systems, this distribution generally comes in the form of a bootable CD or DVD. When the user follows the indications after booting from this media, she ends up with the newest version of her favorite operating system. But could she rebuild the complete distribution using nothing but the binaries and the complete sources provided with this media? Will the binaries produced by the distributor match with the binaries coming from the user compilation? Will the distribution packaged by the distributor be identical to the one produced by the user?

These questions are relevant in practice for the following two reasons. First, any negative answer to the questions above indicates a potential problem with the way the distribution was initially built by the distributor. Such problems could hamper further evolutions of the distribution. Therefore, ensuring that a distribution can be rebuilt from scratch is an additional sanity check that distributors may want to perform systematically. Second, some users have strong needs for building distributions from sources. For instance, as part of thinning a distribution for use on a particular device, recompilation with compilation flags and optimizations appropriate for the device can lead to a smaller or faster custom distribution. Also, users with high security requirements may not trust binaries built by the distributor and elect to build their binaries from sources after performing a security code review over the sources.

5.1 State of the art

This chapter is inter-related with Thinning which we addressed in chapter 4, in particular the state of the art. We are not going to duplicate it here.

Compiling and recompiling from sources is not a new problem in computer science. Moreover, we are interested in rebuilding a Linux distribution from scratch which poses new problems.

5.1.1 Expected issues

Compiler bootstrapping

The issue of bootstrapping a compiler and its solution are well known. The compiler distributions address this problem which is not a Linux distributor one.

Package management bootstrapping

Depending on what tools are choosen to rebuild the distribution, the question of ordering becomes important. The Linux From Scratch documentation [8] chose the lowest level approach, starting with the compiler and a few minimal tools, without the help of a sophisticated package management system. However, a typical distributor will use special tools parts of their package management system such as rpm-rebuilder for Mandriva or buildd for Debian and it automates the manual steps we are going to explain below.

When starting from scratch with the point of view of the sources and of the tools, some considerations must be taken. Let's take the example of the RPM management system. According to the RPM HOWTO, the rpm tools (and consequently all the tools relying on it such as Mandriva's urpmi for example) rely on the existence of core programs to work. Building these tools is a multi-step process.

First, a number of programs need to be available in order to build RPM packages:

- install program from fileutils*.tar.gz
- patch*.tar.gz
- autoconf*.tar.gz
- automake*.tar.gz
- libtool*.tar.gz
- gcc*.tar.gz

Then follow the foundation rpms which are presented in the order of dependency:

- fileutils*.rpm
- grep*.rpm
- gawk*.rpm
- sed*.rpm
- texinfo*.rpm

- zlib*.rpm and zlib-devel
- patch*.rpm
- setup*.rpm
- filesystem*.rpm (one may not want to install this if it affects the /proc directory)
- textutils*.rpm
- glibc-common*.rpm
- basesystem*.rpm
- mktemp*.rpm
- bash*.rpm
- m4*.rpm (autoconf needs this)
- autoconf
- bison
- `binutils >= 2.9.1.0.25`
- gas, as, ld which are in binutils
- shutils - for the 'id' command

A second stage of foundation rpms are needed. After installing the foundation rpms, the next most important rpm is gcc, the order of rpms needed is:

- glibc*.rpm
- binutils*.rpm
- kernel-headers*.rpm
- glibc-devel*.rpm
- gcc*.rpm

Finally, a third stage of rpm:

- popt*.rpm
- rpm*.rpm
- perl*.rpm
- And others, depending on the distribution....

5.1.2 Build systems

As we mentioned above, a Linux distributor will look for high-level tools for rebuilding its distribution from scratch. The package management toolkit is the first place to look for such tools and someone could use it with an ad-hoc process to rebuild a distribution manually. However, a Linux distributor is more interested in a reproductible process which can scale with a number of developers working in parallel, historical data, mass diffusion and short production cycles. This is why the build system is strategically important, because it implements the process of building (or rebuilding) a distribution as most efficiently as possible. We are going to review a number of build systems. We don't pretend to be exhaustive here, but we looked on Debian and a little bit more on Mandriva which is cooperating with EDOS and in the process of adopting a new build system. The reflection which took place will guide the reader in this section.

Let's first review the Debian building process.

Debian

Debian has developed an automated build system [1]. The `buildd` system is used as a distributed, client-server build distribution system. It is usually used in conjunction with auto-builders, which are "slave" hosts which simply check out and attempt to auto-build packages which need to be ported. There is also an email interface to the system, which allows porters to "check out" a source package (usually one which cannot yet be auto-built) and work on it.

A number of tools are described in Debian's developers' reference:

- wanna-build and buildd, for automated builders
- debootstrap
- pbuilder
- sbuild

The `debootstrap` package and script allows you to "bootstrap" a Debian base system into any part of your filesystem. By "base system", we mean the bare minimum of packages required to operate and install the rest of the system. Having a system like this can be useful in many ways. For instance, you can chroot into it if you want to test your build dependencies. Or you can test how your package behaves when installed into a bare base system. Chroot builders use this package; see below.

`pbuilder` constructs a chrooted system, and builds a package inside the chroot. It is very useful for checking if a package's build-dependencies are correct, and for making sure that unnecessary and wrong build dependencies will not exist in the resulting package. A related package is pbuilder-uml, which goes even further by doing the build within a User Mode Linux environment.

`sbuild` is another automated builder. It can use chrooted environments as well. It can be used stand-alone, or as part of a networked, distributed build environment. As the latter, it is part of the system used by porters to build binary packages for all the available architectures. See buildd, Section 5.10.3.3 for more information, and http://buildd.debian.org/ to see the system in action.

Caixa Mágica

Caixa Mágica drives a straight production process. When the tools are ready they are packaged in RPMs and then included in the CD, the packaging process starts by having the source code for the tools in the packaging server file system and the SPECs in the SVN. The SPECs are written individually for each OSS package and have information about how to compile, install and configure it (through pre and post-install scripts). For each software tool that is included in the CD, a source and binary RPM is produced from the SPEC file.

In parallel, outside players like individual contributors and organizations take part in the process by assembling packages of their own. These packages are not included in the official Linux distribution but included in parallel repositories. The best of these packages can later be integrated into future versions of the official branch.

Mandriva also has been looking for new ways to automate its building process. In that perspective, a handful of build systems potentially useful for Mandriva have been studied. We are going to review them now.

Conectiva

The Conectiva build system is centered on a database that stores source data and metadata for revision control. Developers send packages to revision control and submit them for building. The system retrieves the necessary data from the database, builds a source RPM package, stamps it with a serial number (called ”stardate”) and submits it to the autotest box. The autotester lints the specfile, builds the package and installs it on its environment. Should any error occur, the package is rejected and the developer notified. If it passes cleanly, it’s enqueued for buildmaster approval or modification on the final build machine, where it is rebuilt on a strict controlled environment and stored in the binary package repository.

Multi-arch/multi-distro system has been used to build the Unitedlinux-based distribution, the PPC port of Conectiva Linux, several custom distro branches and maintenance of previous releases.

This system is based on the following premises:

- Official packages are to be built in a trusted environment by a trusted user.
- Only extensively tested binaries are to be released (i.e. no massive rebuild prior to release).

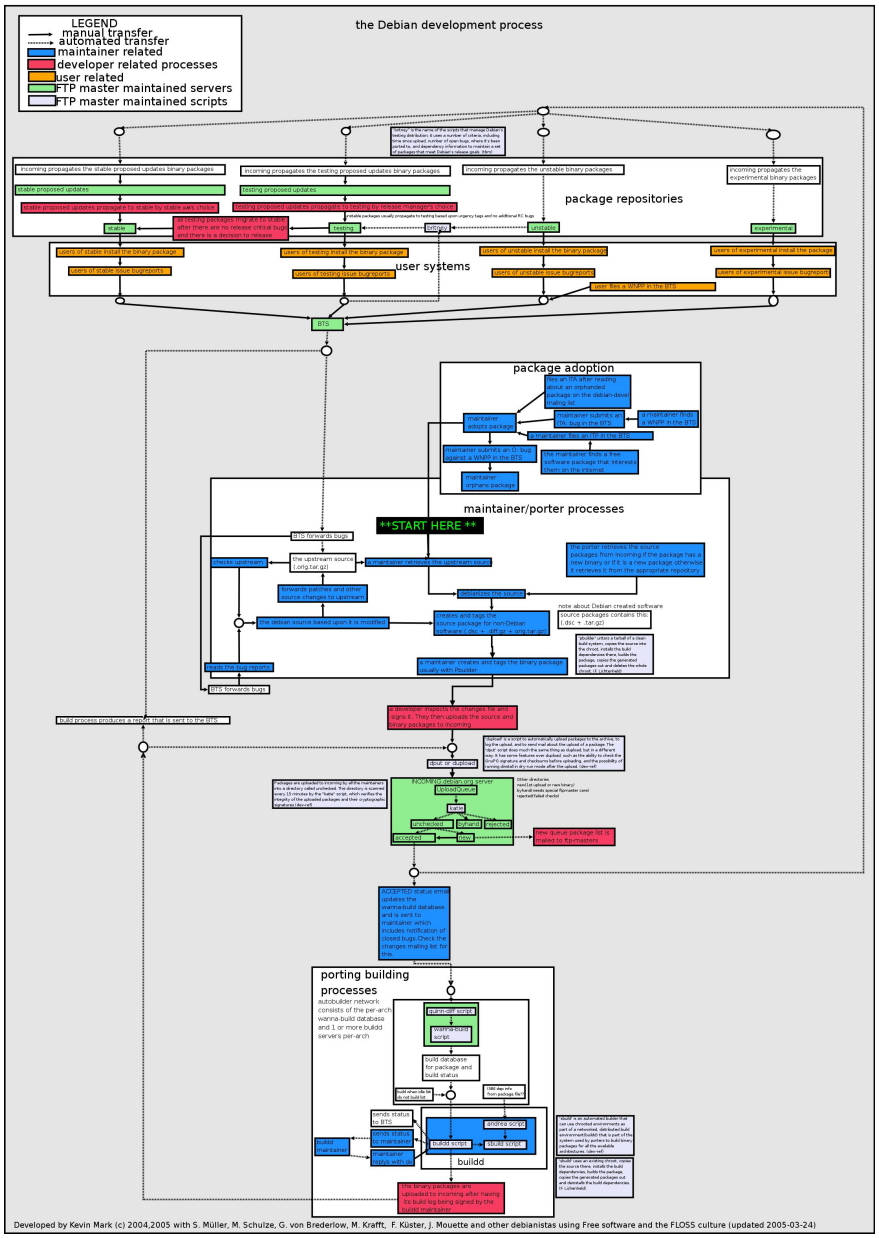


Figure 5.1: Debian build system

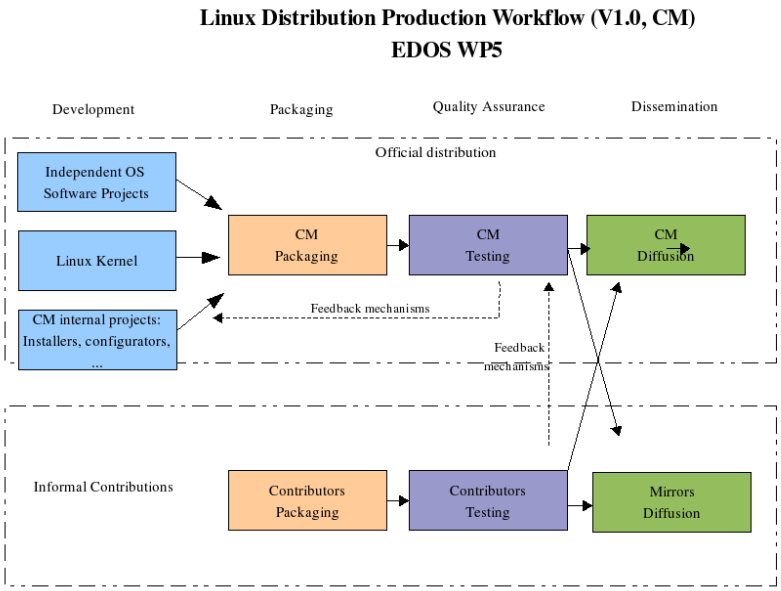


Figure 5.2: Caixa Mágica workflow

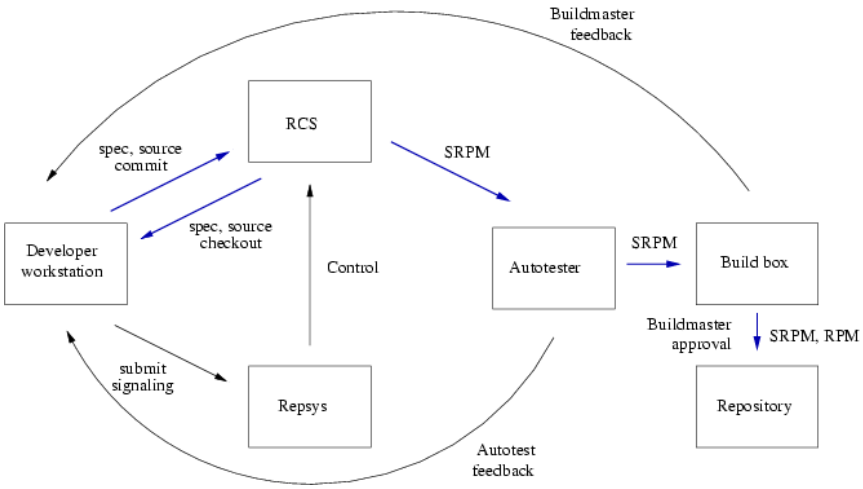


Figure 5.3: Conectiva build system

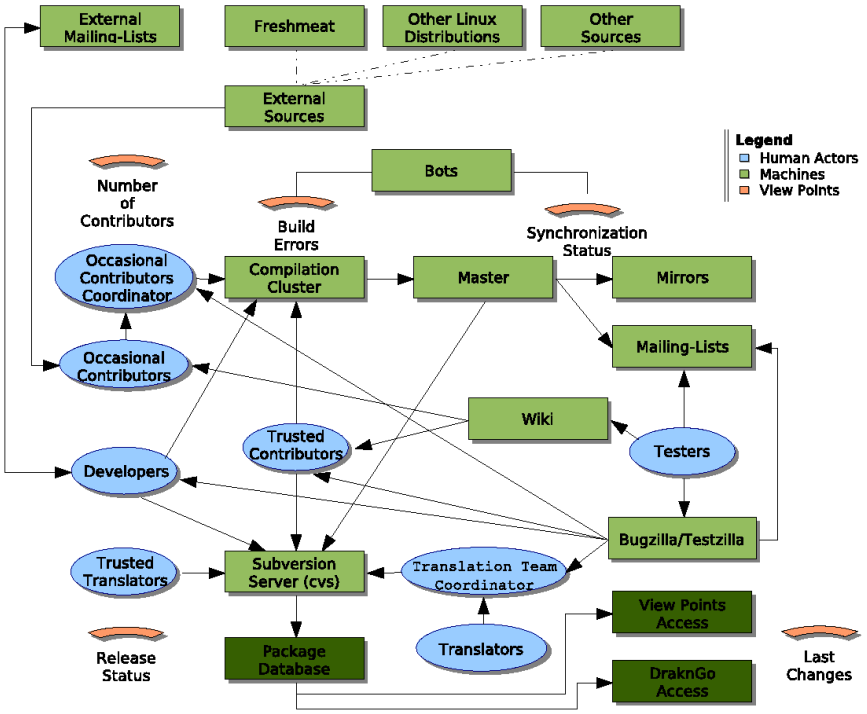


Figure 5.4: Old Mandriva Linux cooker process

- Only policy compliant packages are allowed to enter the distribution.
- Packages must upgrade cleanly.
- Changes initiated from a compromised developer workstation can be audited and rolled back.

Former Mandrivalinux

At Mandriva, the elaboration of a new distribution is an evolutive process happening in the so-called *cooker* distribution and it has been described earlier. Every contributor is free to replace and to add new packages at any time. Traditionnally, a developer works on the cluster to recompile and to test its RPM packages and then she submits them via some scripts such as ftpcooker. After some time needed to reproduce the hdlists, to generate logs and to relocate the packages, they become available. Before the reflection started with EDOS this process could take from 25 to 40 minutes.

The ”Kenobi” architecture used in the Mandrakelinux buildsystem makes a build host shell directly available to the developer. In the build host the developer build the packages and uploads them to Cooker. There is no buildmaster as the

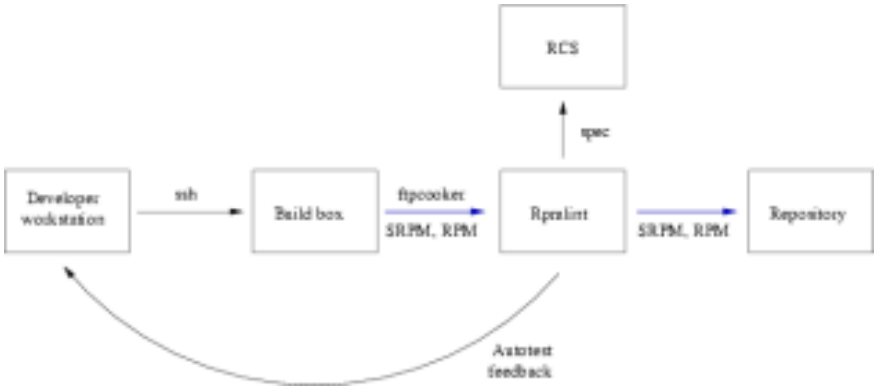


Figure 5.5: Old Mandriva Linux build system

developer-built binary and source packages are sent abroad after Rpmlint checks. Specfiles are put into revision control when it passes Rpmlint.

Mandriva is in the process of moving from this old architecture to the new one described above. Some improvements have already been incorporated. It is expected that the EDOS toolkit will be part of this new system.

EDOS will bring improvement to this for the dependency problems.

LbD

The build environment at csc.warwick.ac.uk uses LbD for rebuilding packages. A number of LbD hosts have been configured that can build packages for the architectures they support (alpha: alpha, i586: i586, ultrasparc: sparc + sparc64, x86_64: i586 + x86_64).

Complete information on LbD can be found on Mandriva's Wiki: http://qa.mandriva.com/twiki/bin/view/Main/LbD.

Its advantages:

- It's already there;
- It rebuilds in a clean (basesystem + rpmbuild) and trusted environment;
- It's a continuous process (unlike other run-once implementations);
- It scales over multiple hosts and architectures;
- It's able to upload to different environments (local, mandriva, plf, etc);
- Outputs are available via web interface;
- Local rebuilding of src.rpms to get TRUE BuildRequires;
- Installation and de-installation of BuildRequires;

- Works with apt, smart and urpmi;
- Compares resulting rpms with whats currently in the repository;

PLF

The PLF build system consist of a set of chroots on different architectures, where maintainers have to build their package manually, then upload the result to the central repository. There is a lead distribution/architecture (cooker/i586) where uploading first is mandatory, and all other distributions/architecture are optional only. Upload enforce a set of mandatory rules, whereas later QA processus scan a broader array of checks.

All the tools used are developed as part of an independant project, Youri. YOURI stands for "Youri Offers an Upload and Repository Infrastucture". It aims to build tools making management of a coherent set of packages easier. Youri's project home page is available at http://youri.zarb.org/ .

5.1.3 Improvements

The production process is in constant evolution and the EDOS project is already a source of improvements for the Linux distributors. Let's take Mandriva as an example to review some of the improvements which took place along the EDOS reflection. Some of these enhancements are directly related to the objectives of EDOS and some others are not directly related but induced by the ongoing reflection. Also some tools and prototypes are being developed by Mandriva's staff, such as iurt.

Overall enhancements

Some enhancements have already been performed such as the performance gain obtained by delegating Input/Output operations to a new machine (Raoh) now in charge of talking to the mirrors. Another improvement comes from the new incremental hdlist generation, which avoids to recompile all the dependencies every time. Finally, the process is now taking around 5 minutes only. After this time, the cooker distribution is updated with the new packages.

Build system

Mandriva is implementing a totally new system. Instead of working manually on Mandriva cluster the package maintainer will use a Subversion repository and she will be able to commit packages remotely from this repository.

The new repository system is based on the original Conectiva's package building and testing system. Current status of the Mandriva repository is TEST. The KDE Mandriva team is the first group to have their packages on this repository.

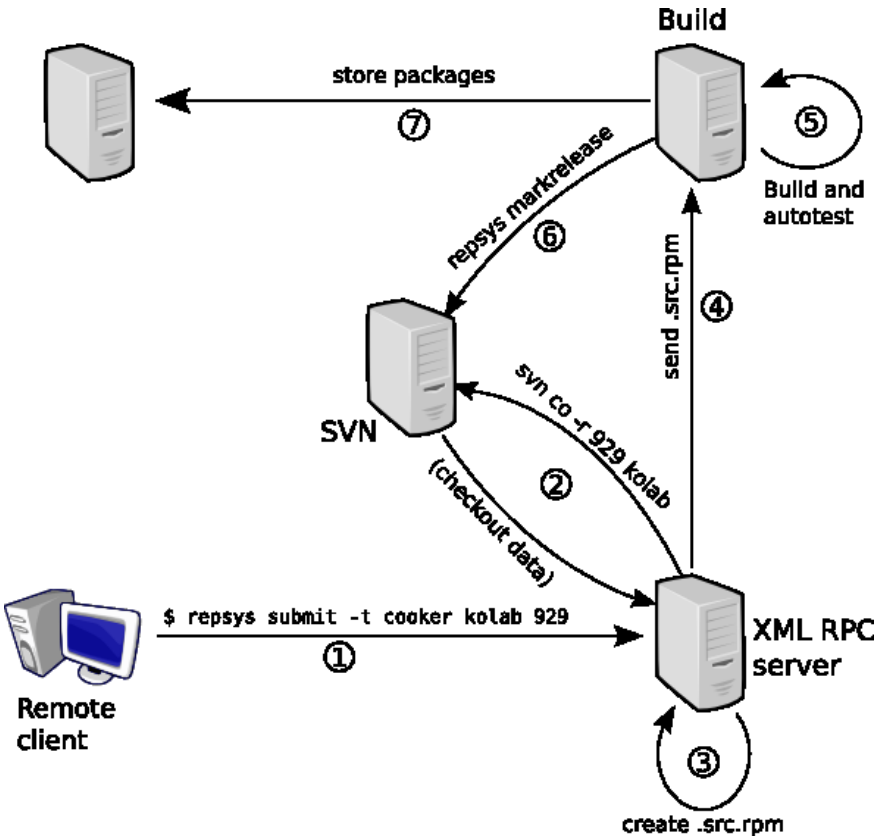


Figure 5.6: Submitting a RPM package to Mandriva with the new build system

Not all infrastructure is available from the start, just the cooker/ tree is open. All the information about how to use this repository is available online [5].

The submission will be made remotely, eventually from the developer's home machine, and with a single command. The diagram below shows how the submission process happens internally.

1. submit: the user issues the submit command via repsys. If authentication and authorization is successful, the submit command is sent to the XML RPC server;

2. svn checkout: the XML RPC server checks out the requested revision of the package from SVN (the SVN layout is explained in RepositorySystem)

3. rpm -bs: the XML RPC server creates a .src.rpm file with the data checked out from SVN. A generic script/program can optionally be called prior to "rpm -bs" to, for example, change the release number in the spec file (we call this "rebrand");

4. send away: the XML RPC server stores or sends the .src.rpm file somewhere to be picked up by the build daemon;
5. build: the build daemon/machine grabs the .src.rpm, builds it and tests it and the resulting binaries;
6. markrelease: if the build succeeded, and the package passed all tests, then the build daemon/machine creates a new entry in the ”release” directory in SVN for this package;
7. ready: the .src.rpm file and the binaries just built are stored/sent somewhere, ready to be, for example, signed and uploaded to a mirror.

iurt

Iurt is a recompilation bot which monitors lists of packages of different architectures and recompiles them in a separated clean chroot each time it is needed.

Iurt is monitoring the source and binary packages from the various repositories and media, and recompiles them if needed. There are four different usages:

- `iurt --distro cooker --config_help` to display current values from config files, which will be used
- `iurt --distro cooker --chroot --arch i586` to create or update the tarball for the chroot
- `iurt --distro cooker --rebuild i586 cooker /path/to/foo.src.rpm` to build the package foo in the chroot
- `iurt --distro cooker --arch i586 --media bar` to rebuild all non up-to-date SRPMS for the media bar

Iurt first checks if another iurt is running, then scans the packages in the repository. After that it computes the list of packages which need to be recompiled, create or verify the chroot tar which will be used to recompile the packages. It then loops on the packages to compile, reinstalling a new chroot each time, then the build dependencies, compiling the package, trying to install the resulting binaries, and copying them to their final destination.

For running it, you’ll need at least: perl-File-NCopy perl-Data-Dump perl-MIME-tools (you can use urpmi to install them). Non packaged script: install-chroot-tar.sh (http://cvs.mandriva.com/cgi-bin/cvsweb.cgi/soft/rpm-rebuilder/)

5.2 Current limitations

We can identify two issues dealing with package rebuilding and they happen in well-defined occurrences:

- when rebuilding, for a given distribution, a single package or a small set of interdependent packages, typically while performing a security fix;
- when a user proceeds to rebuild the distribution from the sources provided, using only the tools provided by the distribution, such as the binaries for the compiler `gcc` and the C library `libc`.

5.2.1 Rebuilding a single package from scratch

In order to make explicit the difficulties encountered by the distributors and the maintainers, we are going to present some scenarios. Each one will showcase a particular issue which we will discuss further. The first two scenarios deal with the first problem, to rebuild for a given distribution a single package or a small set of interdependent packages.

Case study 1: Compilation of Ekiga (the new gnomemeeting) Ekiga depends on the H323 and SIP libraries. In order to build a new version of Ekiga, we need the latest version of these libraries, and we need to compile these first. Then, we can compile Ekiga. But a number of problems arise:

1. the order of compilation must be respected (this is solved as of right now by a script);
2. the older versions of the libraries must be removed from the compilation environment (if they were present);
3. new versions of the libraries are added to the compilation environment.

The first issue is easily solved by a script and there is no collateral effect. However, the next two issues demonstrate a contamination of the compilation environment. For example, if a developer wants to compile her package, she may be affected by the missing libraries. But the most common situation appears when new libraries have been added to the compilation environment. Because the compilation is automated, and it relies on `configure` scripts, some compilation options can inadvertently be turned on, therefore introducing new effective dependencies which were not accounted for, nor even anticipated.

The lesson from this case is that a developer needs to rely on a *clean* compilation environment. A *clean* compilation environment is to be understood as an environment which is composed of the minimal or base set of packages for the distribution (see the chapter on thinning).

Case study 2: Compilation of Curl Curl is a tool to transfer data from or to a server providing the most commonly used protocols such as HTTP and FTP. This case is similar to the previous one with one difference. The RPM packager can decide to include the library `libnet` in the compilation environment or not. We have two possible outcomes:

1. If `libnet` is available in the compilation environment, the `configure` script will detect it and Curl will compile with the extra ability to resolve domain names, for example.

2. If, on the contrary, `libnet` is not available, Curl will not be compiled with this extra feature.

The developer in charge of the package hopefully knows everything about this and she will then include (or not) the appropriate dependency in the RPM meta-data.

Curl comes with a test suite, hence the developer can run some diagnostics on the Curl binaries. However, the test suite is built with the same options as the main Curl binary. Therefore, if `libnet` is missing from the compilation environement the tests for the functionalities depending on `libnet` will not be compiled either. This is not an error, but it shows that a developer can not rely on the test suite, however complete it is, to guess back the compilation options which were activated at compilation time.

5.2.2 Rebuilding the complete distribution from scratch

Compiling the whole distribution from scratch is more complex. In fact, the same issues appear but also the ordering of packages for the compilation needs to be defined. While some order is implicitely defined by the dependencies between packages there are other dependencies such as the existence of a compiler or a set of libraries.

To recompile the distribution rpm-rebuider can be used. A sample configuration file (/.rpm-rebuilder) contains the following options:

```
DISTRIB=/home/cooker
RB_LABORATORY=$DISTRIB/rebuild/main
SRPMS_DIRS=$DISTRIB/SRPMS/main
RPM_TOP_DIR=$RB_LABORATORY/rpm
RPM_DEST_DIR=$DISTRIB/RPMS
RPM_DEBUG_DEST_DIR=$DISTRIB/debug
RPMS_DIRS=$RPM_DEST_DIR
INSTALL_RPMS=1
ORDERER=$HOME/bin/orderer.main
```

The use of `rpm-rebuilder` to rebuild the distribution from scratch raises several issues. First, the release number must be auto-incremented, otherwise the installation of the package will try to replace the current one — and this will fail, unless forced. To compare the binaries, it is necessary to keep track of the different versions of the (same) packages, and also of the binaries they contain. Second, to manage the boostraping it is necessary to use an orderer. compute-compile-order generates the list, which needs to be cleaned up from time to time. Finally, the binaries obtained by recompiling will sometimes always be different from the original

binaries because the binaries include time stamps, for example as generated by C macros such as `__DATE__` and `__TIME__`.

5.3 Ongoing work

A tool (`iurt`) already exists but it does not solve the entire problem: some steps still need to be performed manually. It should be evaluated in order to determine if it can be improved or if a better solution exist.

Currently, `iurt` manages the compilation environment in the form of a tarball which it unpacks, uses as chroot, and then removes, repeating this process for each one of the packages to rebuild. An alternative algorithm is currently being explored to gain time and resources. The new `iurt` will use unionfs to access the minimal compilation environment without the need to unpack a compressed file.

In current practice, as we can see in case 1, the compilation order is specified by scripts, hand-written and tailored to a specific package or set of packages. If the process of determining the compilation order were to be automated, the algorithm should be generic and the data related to the compilation order, for example, should be extracted from the script and stored in another fashion. This data could eventually be part of the spec file, for example.

The compilation robot must be able to retrieve the correct information on the required compilation environment for a specific package in order to re-create it. But it needs also the information about the compilation order when multiple packages are involved.

Chapter 6

Dependency management

In this chapter, we introduce the terminology that will be used to describe the entities related to the server-side maintenance of package repositories, and we identify and define the main server-side desirable properties that need to be checked when maintaining a repository, as far as package dependencies are concerned.

The formal definition of these properties is essential to tackle properly the algorithmic challenge that they pose.

6.1 Basic definitions

Every package management system [14, 6] takes into account the interrelationships among packages (to different extents). We will call these relationships *requirements*. Several kinds of requirements can be considered. The most common one is a *dependency* requirement: in order to install package P_1, it is necessary that package P_2 is installed as well. Less often, we find *conflict* requirements: package P_1 cannot coexist with package P_2.

Some package management systems specialize these basic types of requirements by allowing to specify the *timeframe* during which the requirement must be satisfied. For example, it is customary to be able to express *pre-dependencies*, a kind of dependency stating that a package P_1 needs package P_2 to be present on the system *before* P_1 can be installed [14].

In the following, we assume the distribution and the architecture are fixed. We will identify packages, which are archive files containing metadata and installation scripts, with pairs of a unit and a version.

Definition 1 (Package, unit) *A* package *is a pair (u, v) where u is a unit and v is a version. Units are arbitrary strings, and we assume that versions are non-negative integers.*

While the ordering over version strings as used in common OSS distributions is not discrete (i.e., for any two version strings v_1 and v_2 such that $v_1 < v_2$, there exists v_3 such that $v_1 < v_3 < v_2$), taking integers as version numbers is justified for two

reasons. First, any given repository will have a finite number of packages. Second, only packages with the same unit will be compared.

For instance, if our Debian repository contains the versions `2.15-6`, `2.16.1-cvs20051117-1` and `2.16.1cvs20051206-1` of the unit `binutils`, we may encode these versions respectively as 0,1 and 2, giving the packages $(\texttt{binutils}, 0)$, $(\texttt{binutils}, 1)$, and $(\texttt{binutils}, 2)$.

Definition 2 (Repository) *A* repository *is a tuple $R = (P, D, C)$ where P is a set of packages, $D : P \to \mathscr{P}(\mathscr{P}(P))$ is the dependency function*[1]*, and $C \subseteq P \times P$ is the conflict relation. The repository must satisfy the following conditions:*

- *The relation C is symmetric, i.e., $(\pi_1, \pi_2) \in C$ if and only if $(\pi_2, \pi_1) \in C$ for all $\pi_1, \pi_2 \in P$.*
- *Two packages with the same unit but different versions conflict*[2]*, that is, if $\pi_1 = (u, v_1)$ and $\pi_2 = (u, v_2)$ with $v_1 \neq v_2$, then $(\pi_1, \pi_2) \in C$.*

In a repository $R = (P, D, C)$, the dependencies of each package p are given by $D(p) = \{d_1, \ldots, d_k\}$ which is a set of sets of packages, interpreted as follows. If p is to be installed, then all its k dependencies must be satisfied. For d_i to be satisfied, at least one of the packages of d_i must be available. In particular, if one of the d_i is the empty set, it will never be satisfied, and the package p is not installable.

Example 1 *Let $R = (P, D, C)$ be the repository given by*

$$
\begin{aligned}
&P = \{a, b, c, d, e, f, g, h, i, j\} \\
&D(a) = \big\{\{b\}, \{c, d\}, \{d, e\}, \{d, f\}\big\} \\
&D(b) = \big\{\{g\}\big\} \quad D(c) = \big\{\{g, h, i\}\big\} \quad D(d) = \big\{\{h, i\}\big\} \\
&D(e) = D(f) = \big\{\{j\}\big\} \\
&C = \{(c, e), (e, c), (e, i), (i, e), (g, h), (h, g)\}
\end{aligned}
$$

where $a = (u_a, 0)$, $b = (u_b, 0)$, $c = (u_c, 0)$ and so on. The repository R is represented in figure 6.1. For the package a to be installed, the following packages must be installed: b, either c or d, either d or e, and either d or f. Packages c and e, e and i, and g and h cannot be installed at the same time.

In computer science, dependencies are usually *conjunctive*, that is they are of the form

$$a \to b_1 \wedge b_2 \wedge \cdots \wedge b_s$$

where a is the target and b_1, b_2, ... are its prerequisites. This is the case in `make` files, where all the dependencies of a target must be built before building the target. Such dependency information can be represented by a directed graph, and

[1] We write $\mathscr{P}(X)$ for the set of subsets of X.

[2] This requirement is present in some package management systems, notably Debian's, but not all. For instance, RPM-based distributions allow simultaneous installation of several versions of the same unit, at least in principle.

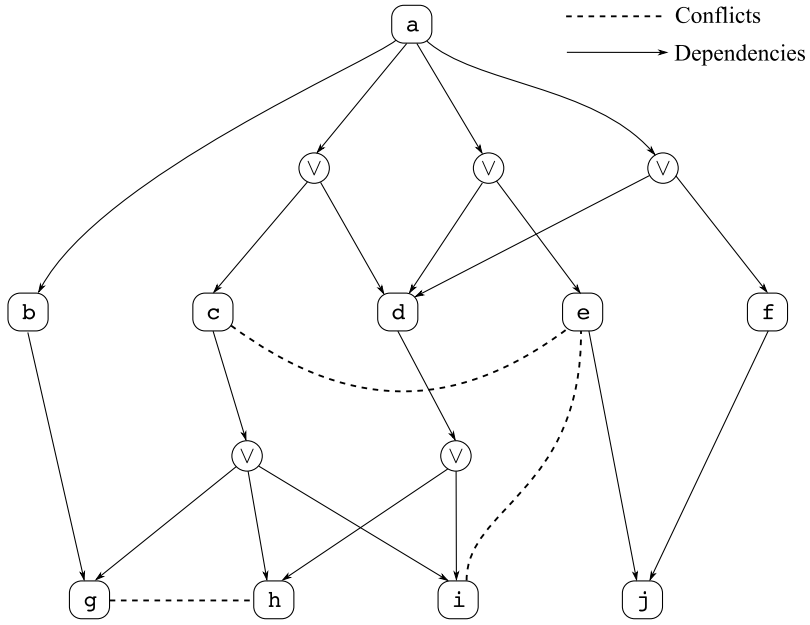


Figure 6.1: The repository of example 1.

dependencies can be solved by the well-known topological sort algorithm. Our dependencies are of a more complex kind, which we name *disjunctive* dependencies. Their general form is a conjunction of disjunctions:

$$a \rightarrow (b_1^1 \vee \cdots \vee b_1^{r_1}) \wedge \cdots \wedge (b_s^1 \vee \cdots \vee b_s^{r_s}). \tag{6.1}$$

For a to be installed, each term of the right-hand side of the implication 6.1 must be satisfied. In turn, the term $b_i^1 \vee \cdots \vee b_i^{r_i}$ when $1 \leq i \leq s$ is satisfied when at least one of the b_i^j with $1 \leq j \leq r_i$ is satisfied. If a is a package in our repository, we therefore have

$$D(a) = \{\{b_1^1, \ldots, b_1^{r_1}\}, \cdots, \{b_s^1, \ldots, b_s^{r_s}\}\}.$$

In particular, if one of the terms is empty (if $\varnothing \in D(a)$), then a cannot be satisfied. This side-effect is useful for modeling repositories containing packages mentioning another package b that is not in that repository. Such a situation may occur because of an error in the metadata, because the package b has been removed, or b is in another repository, maybe for licensing reasons.

Concerning the relation C, two packages $\pi_1 = (u_1, v_1), \pi_2 = (u_2, v_2) \in P$ conflict when $(\pi_1, \pi_2) \in C$. Since conflicts are a function of presence and not of installation order, the relation C is symmetric.

Definition 3 (Installation) *An* installation *of a repository $R = (P, D, C)$ is a subset of P, giving the set of packages installed on a system. An installation is* healthy

when the following conditions hold:

- ***Abundance:*** *Every package has what it needs. Formally, for every $\pi \in I$, and for every dependency $d \in D(\pi)$ we have $I \cap d \neq \varnothing$.*
- ***Peace:*** *No two packages conflict. Formally, $(I \times I) \cap C = \varnothing$.*

Definition 4 (Installability and co-installability) *A package π of a repository R is* installable *if there exists a healthy installation I such that $\pi \in I$. Similarly, a set of packages Π of R is* co-installable *if there exists a healthy installation I such that $\Pi \subseteq I$.*

Note that because of conflicts, every member of a set $X \subseteq P$ may be installable without the set X being co-installable.

Example 2 *Assume a depends on b, c depends on d, and c and d conflict. Then, the set $\{a, b\}$ is not co-installable, despite each of a and b being installable and not conflicting directly.*

Definition 5 (Maximal co-installability) *A set X of co-installable packages of a repository R is* maximal *if there is no other co-installable subset X' of R that strictly contains X. We write $\mathrm{maxco}(R)$ for the family of all maximal co-installable subsets of R.*

Definition 6 (Dependency closure) *The* dependency closure $\Delta(\Pi)$ *of a set of package Π of a repository R is the smallest set of packages included in R that contains Π and is closed under the* immediate dependency *function $\overline{D} : \mathscr{P}(P) \to \mathscr{P}(P)$ defined as*

$$\overline{D}(\Pi) = \bigcup_{\substack{\pi \in \Pi \\ d \in D(\pi)}} d.$$

In simpler words, $\Delta(\Pi)$ contains Π, then all packages that appear as immediate dependencies of Π, then all packages that appear as immediate dependencies of immediate dependencies of Π, and so on. Since the domain of $\overline{D}$ is a complete lattice, and $\overline{D}$ is clearly a continuous function, we immediately get (by Tarski's theorem) that such a smallest set exists and can be actually computed as follows:

Proposition 1 *The dependency closure $\Delta(\Pi)$ of Π is:*

$$\Delta(\Pi) = \bigcup_{n \geq 0} \overline{D}^n(\Pi).$$

The notion of dependency closure is useful to extract the part of a repository that pertains to a package or to a set of packages.

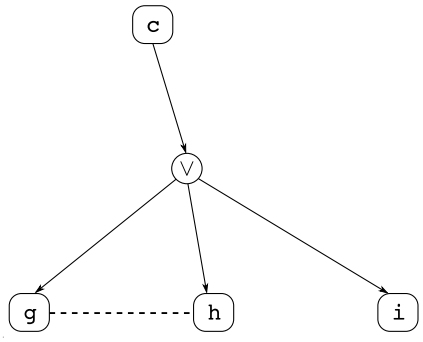


Figure 6.2: The subrepository generated by package c. The dependency closure is $\{c, g, h, i\}$.

Definition 7 (Generated subrepository) *Let $R = (P, D, C)$ be a repository and $\Pi \subseteq P$ be a set of packages. The* subrepository generated by Π *is the repository $R|_\Pi = (P', D', C')$ whose set of packages is the dependency closure of Π and whose dependency and conflict relations are those of R restricted to that set of packages. More formally we have $P' = \Delta(\Pi)$, $D' : P' \to \mathscr{P}(\mathscr{P}(P')), \pi \mapsto \{d \cap P' \mid d \in D(\pi)\}$ and $C' = C \cap (P' \times P')$.*

Figure 6.2 shows the subrepository generated by the package c of example 1. The dependency closure of c is the set of package nodes of that subrepository. A larger, real-world example is shown in figure 6.3.

We then have the following property, which allows to consider only the relevant subrepositories when answering questions of installability.

Proposition 2 (Completeness of subrepositories) *A package π is installable w.r.t. R if and only if it is installable w.r.t. $R|_\pi$. (Similarly for co-installability.)*

The Dependency Closure definition actually corresponds to the definition of *Island of Packages* used in the context of WP5. An *Island of Packages*, in fact, is defined as the *dependency closure on a package or a set of packages, that includes all the packages required to install them*.

6.2 Relevant desirable properties of a package repository

The task of maintaining a package repository is difficult: the maintainance team must monitor the evolution of thousand of packages over time, and address the error reports coming from different sources (users, QA teams, developers, etc.). It is desirable to automate as much of this work as possible. Our medium-term goal is to build tools that help distribution maintainers track dependency-related problems in package repositories. We detail here some of the desirable properties

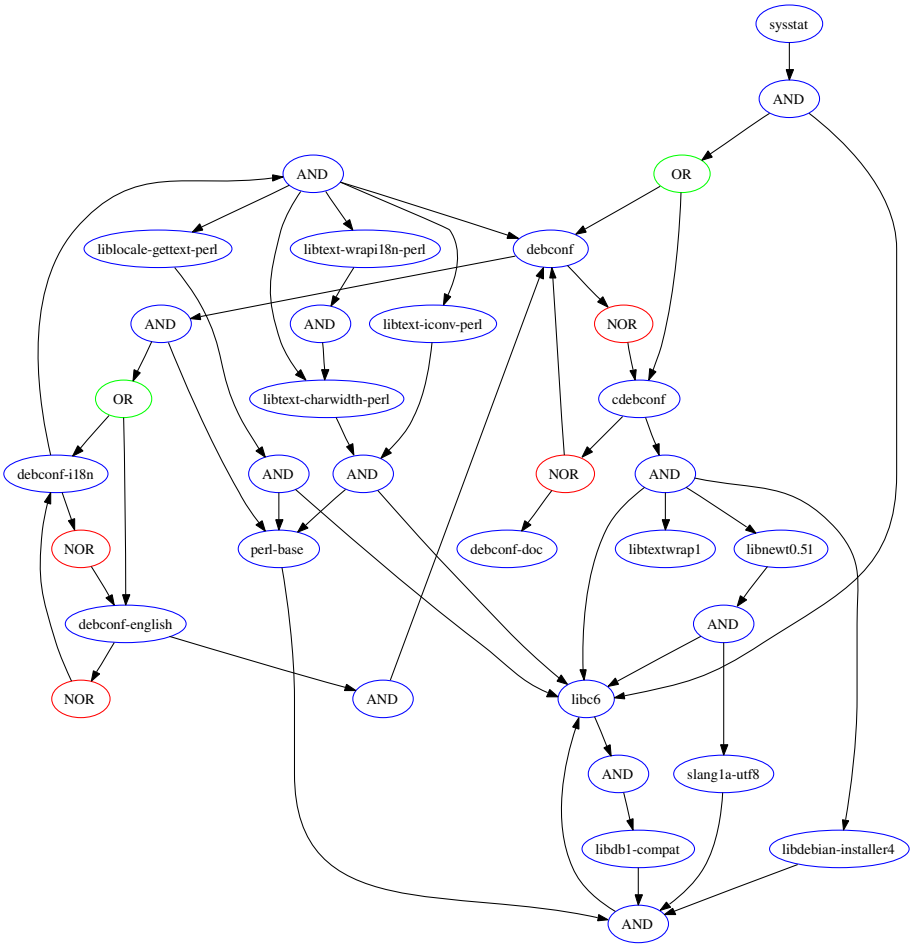


Figure 6.3: The subrepository generated by the package `sysstat` in Debian stable on 2005-12-13, viewed as a boolean graph. Version numbers have been omitted for brevity.

of a repository. The first is *history-free*, in that it applies to a given state of a repository.

Being trimmed We say that a repository R is *trimmed* when every package of R is installable w.r.t. R. The intuition behind this terminology is that a non-trimmed repository contains packages that cannot be installed in any configuration. We call those packages *broken*. They behave as if they were not part of the repository. It is obviously desirable that at any point in time, a repository is trimmed, that is, contains no broken packages.

The next properties are *history-sensitive*, meaning that they take into account the evolution of the repository over time. Due to this dependency on time, the precise formulation of these properties is delicate. Just like history-free properties are relevant to users who install a distribution from scratch, history-sensitive properties are relevant to users who upgrade an existing installation.

Monotonicity Let R_t be the repository at time t and consider a coinstallable set of packages C_t. Some users can actually have packages C_t installed simultaneously on their system. These users have the possibility of installing additional packages from R_t, resulting in a coinstallable set of packages C'_t. These users can reasonably expect that they will be able to do so (extend C_t into C'_t) at any future time t', using the repository $R_{t'}$, which, being *newer*, is supposed to be *better* than the old R_t.

Of course, users are ready to accept that in $R_{t'}$ they will not get exactly C'_t, but possibly $C'_{t'}$, where some packages were updated to a greater version, and some others have been replaced as the result of splitting into smaller packages or grouping into larger ones. But, clearly, it is not acceptable to evolve R_t into $R_{t'}$ if R_t allows to install, say, `apache` together with `squid`, while $R_{t'}$ does not.

We say that a repository history line is *monotone* if the *freedom* of a user to install packages is a monotone function of time. Writing $F(x, R)$ for the set of possible package sets in R that are a possible replacement of package x according to the metadata, monotonicity can be formally expressed as

$$\mathrm{Mon}(R) = \forall t < t'.\, \forall P \in \mathrm{Con}(R_t).\, \exists Q \in \mathrm{Con}(R_{t'}).\, \forall x \in P.\, Q \cap F(x, R_{t'}) \neq \varnothing$$

Upgradeability Another reasonable expectation of the user is to be able to upgrade a previously installed package to the most recent version (or even any more recent version) of this package that was added to the repository since her latest installation. She is ready to accept that this upgrade will force the installation of some new packages, the upgrade of some other packages, and the replacement of some sets of package by other sets of packages, as the result of the reorganization of the structure of the packages. However, she cannot accept that the upgrade of a package forces the complete removal of other previously installed packages that she uses.

In other terms, the evolution of a repository respects the upgradeability property if all upgrades of individual packages can be performed without loss of functionality: all packages removed as part of such an upgrade must be compensated by the installation of other packages of equivalent functionality. The notion of "equivalent functionality" needs to be indicated in the metadata of the packages, such as for instance the "Replaces" clauses in Debian's package metadata.

We remark that these properties are *not* interdefinable. We give here a proof of this assertion by exhibiting example repositories showing this independence of the properties. For the first two cases, consider three repositories R_1, R_2, R_3 whose sets of packages are $P_1 = \{(a,1),(b,1),(c,1)\}$, $P_2 = \{(a,1),(b,1)\}$, $P_3 = \{(a,1),(a,2),(b,1)\}$ with no conflicts nor dependencies among the version 1 packages and a conflict among $(a,2)$ and $(b,1)$. Notice that at each moment t in time, R_t is trimmed.

1. A repository that stays trimmed over a period of time is not necessarily monotone, nor upgradeable. Since $(c,1)$ disappears between times 1 and 2, this step in the evolution does not preserve monotonicity. Since $(a,2)$ has a new conflict (namely with $(b,1)$) in R_3, the evolution from R_2 to R_3 does not preserve upgradeability.

2. A repository that stays trimmed over a period of time and evolves in a monotone fashion is not necessarily upgradeable. The evolution from R_2 to R_3 above is monotone, each of R_2 and R_3 is trimmed, but we fail upgradeability because there is no way of going from $\{(a,1),(b,1)\}$ to $\{(a,2),(b,1)\}$ because of the conflict.

3. A repository that stays trimmed over a period of time and is upgradeable is not necessarily monotone.

   Consider repositories R_1 and R_2 with $P_1 = \{(a,1),(b,1)\}$ and $P_2 = \{(a,2),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are isolated packages, while $(a,2)$ conflicts with $(b,1)$. Now, a user having installed all of R_1 and really willing to get $(a,2)$ can do it, but at the price of giving up $(b,1)$. This evolution of the repository is therefore upgradeable but not monotone.

4. A repository that evolves in a monotone and upgradeable fashion is not necessarily trimmed at any time: indeed, the monotonicity and upgradeability property only speak of *consistent* subsets of a repository, that cannot contain, by definition, any broken packages.

   Consider for example repositories R_1, R_2 with $P_1 = \{(a,1)\}$, $P_2 = \{(a,1),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are broken because they depend on a missing package $(c,1)$. Here, the evolution of R_1 to R_2 is trivially monotone and upgradeable, because there is *no* consistent subset of R_1 and R_2, and both R_1 and R_2 are not trimmed because they contain broken packages.

The examples above to prove that the three properties are actually independent may seem contrived, but are simplifications of real-world scenarii. For instance, example 3 can actually happen in the evolution of real repositories, when for some reason the new version of a set of interrelated packages is only partially migrated to the repository. Many packages are split into several packages to isolate architecture-independent files, as in the Debian packages `swi-prolog` and `swi-prolog-doc`. When performing this split, it is quite natural to add a conflict in `swi-prolog-doc` against old, non-splitted versions of `swi-prolog`. If the new version of `swi-prolog-doc` slips into a real repository before the new, splitted version of `swi-prolog`, we are exactly in situation number 3 above.

Package developers seem aware of some of these issues: they actually do their best to ensure monotonicity and upgradeability by trying to reduce as much as possible the usage of conflicts, and sometime resorting to naming conventions for the packages when a radical change in the package happens, like in the case of `xserver-common` vs. `xserver-common-v3` in Debian, as can be seen in the dependencies for `xserver-common`.

```
Package: xserver-common
Conflicts: xbase (<< 3.3.2.3a-2), xsun-utils, xbase-clients (<< 3.3.6-1),
  suidmanager (<< 0.50), configlet (<= 0.9.22),
  xserver-3dlabs (<< 3.3.6-35), xserver-8514 (<< 3.3.6-35),
  xserver-agx (<< 3.3.6-35), xserver-common-v3 (<< 3.3.6-35),
  xserver-fbdev (<< 3.3.6-35), xserver-i128 (<< 3.3.6-35),
  xserver-mach32 (<< 3.3.6-35), xserver-mach64 (<< 3.3.6-35),
  xserver-mach8 (<< 3.3.6-35), xserver-mono (<< 3.3.6-35),
  xserver-p9000 (<< 3.3.6-35), xserver-s3(<< 3.3.6-35),
  xserver-s3v (<< 3.3.6-35), xserver-svga (<< 3.3.6-35),
  xserver-tga (<< 3.3.6-35), xserver-vga16 (<< 3.3.6-35),
  xserver-w32 (<< 3.3.6-35), xserver-xsun (<< 3.3.6-35),
  xserver-xsun-mono (<< 3.3.6-35), xserver-xsun24 (<< 3.3.6-35),
  xserver-rage128, xserver-sis
```

6.3 Algorithmic considerations

Our research objective within the EDOS project is to formally define the desirable properties of repositories stated above (and possibly other properties that will appear useful), and to develop efficient algorithms to check these properties automatically.

It is really not evident that any of these problems are actually tractable in practice: due to the rich language allowed to describe package dependencies in the mainstream FOSS distributions, even the simplest problems (checking installability of a single package) may involve verifications over a large number of other packages. During our first investigations of these problems, we have indeed already proven the following complexity result (see Deliverable D2.1).

Theorem 1 (Package installability is an NP-complete problem) *Checking whether a single package P can be installed, given a repository R, is NP-complete.*

Nevertheless, this strong limiting result does not mean that we will not be able to decide installability and the other problems in practice: the actual instances of these problems, as found in real repositories, could be quite simple in the average.

In particular, the converse of the reduction used for the NP-completeness proof leads to an effective way of deciding package installability, that we will detail in a later chapter.

We are now focusing our attention on the two time-dependent desirable properties for the repositories, which are, algorithmically speaking, much harder.

6.3.1 Encoding the Installability problem as a SAT problem

The formalization of installability provided above leads quite naturally to an encoding as a boolean satisfiability problem. We define a propositional variable I_u^v for every package (u, v) in the repository R, indicating that unit u is installed in version v. We denote for unit u by R_u the set of versions v such that $(u, v) \in R$.

We then build a boolean formula R_s stating that package (u, v) is installable as a conjunction of the following boolean formulas:

At most one version per unit: For every unit u in the repository:

$$\bigwedge_{\substack{v_1, v_2 \in R_u \\ v_1 \neq v_2}} \neg(I_u^{v_1} \wedge I_u^{v_2})$$

Constraints: If R contains a dependency for (u, v) of the form

$$\begin{aligned} &Depends : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \\ &\wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}). \end{aligned}$$

we introduce the formula

$$\begin{aligned} &I_u^v \Rightarrow (\textstyle\bigvee_{w\ op_1^1 v_1^1} I_{u_1}^w \vee \cdots \vee \bigvee_{w\ op_1^{r_1} v_1^{r_1}} I_{u_{r_1}}^w) \\ &\wedge \ldots \wedge (\textstyle\bigvee_{w\ op_s^1 v_s^1} I_{u_s}^w \vee \cdots \vee \bigvee_{w\ op_s^{r_s} v_s^{r_s}} I_{u_{r_s}}^w) \end{aligned}$$

If R contains a conflict for (u, v) of the form

$$\begin{aligned} &Conflicts : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \\ &\wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}). \end{aligned}$$

we introduce the formula

$$\begin{aligned} &I_u^v \Rightarrow (\textstyle\bigwedge_{w\ op_1^1 v_1^1} \neg I_{u_1}^w \vee \cdots \vee \bigwedge_{w\ op_1^{r_1} v_1^{r_1}} \neg I_{u_{r_1}}^w) \\ &\wedge \ldots \wedge (\textstyle\bigwedge_{w\ op_s^1 v_s^1} \neg I_{u_s}^w \vee \cdots \vee \bigwedge_{w\ op_s^{r_s} v_s^{r_s}} \neg I_{u_{r_s}}^w) \end{aligned}$$

Proposition 3 *A package (u, v) is installable in the repository R if and only if the boolean formula $R_s \wedge I_u^v$ is satisfiable.*

Satisfiability of this formula can be checked by a SAT solver.

6.3.2 Encoding the Installability problem as a CP problem

We can also formulate the installability problem for a given package in a Debian repository R as a CP problem over finite domains, but in this case we must start from the repository *before* expanding the version relationships.

To simplify the problem by getting rid of the inessential details related to version comparison algorithms in DEB or RPM formats, we first preprocess the repository and replace version strings by integer as follows: for each unit u, collect all of its mentioned version strings v, and order them accordingly to the appropriate, format specific, comparison algorithm; then replace each occurrence of $u\ op\ v$ by $u\ opn_v$, where n_v is the position of v in the increasingly ordered sequence of versions of u. In other terms, we simply *project over an initial segment of the integers starting at 1* the order structure of the versions of each package. This does not change the nature of the constaint problem, but reduces it to a problem over the Integer domain, for which solvers are more easily available.

We then build a constraint satisfaction problem over a finite domain by constructing a set of constraints R_c out of R as follows:

Variables and domains: For each unit u in the repository R, we introduce a finite domain variable, with domain equal to the set of available versions of the unit present in the repository, plus one special value 0 denoting the fact that no version of the unit is installed. We add the constraint $X_u \in \{0, v_1, \ldots, v_k\}$ to R_c.

Constraints We add constraints to R_c that encode the dependency information associated to each package $\pi = (u, v) \in R$ as follows. If R contains a dependency for $\pi = (u, v)$ of the form

$$Depends : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \\ \wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}).$$

we introduce the constraint

$$(X_u = v) \Rightarrow (X_{u_1^1}\ op_1^1\ v_1^1 \vee \cdots \vee X_{u_1^{r_1}}\ op_1^{r_1}\ v_1^{r_1}) \\ \wedge \cdots \wedge (X_{u_s^1}\ op_s^1\ v_s^1 \vee \cdots \vee X_{u_s^{r_s}}\ op_s^{r_s}\ v_s^{r_s})$$

If R contains a conflict for $\pi = (u, v)$ of the form

$$Conflicts : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \\ \wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}).$$

we introduce the constraint

$$(X_u = v) \Rightarrow \\ (X_{u_1^1}\ compl(op_1^1)\ v_1^1 \wedge \cdots \wedge X_{u_1^{r_1}}\ compl(op_1^{r_1})\ v_1^{r_1}) \\ \vee \cdots \wedge (X_{u_s^1}\ compl(op_s^1)\ v_s^1 \wedge \cdots \wedge X_{u_s^{r_s}}\ compl(op_s^{r_s})\ v_s^{r_s})$$

where $compl(op)$ is the operation opposite to op (e.g., $compl(>>)$ is $<=$, etc.)

Notice that, in the encoding above, if we encounter a package name with no version constraint (so we find just `u` instead of `u >> 3`, for example), we simply produce $X_u > 0$ as the encoding. It is now possible to prove the following:

Proposition 4 *A package $\pi = (u, v)$ is installable in the repository R if and only if the constraint $X_u == v$ is compatible with R_c.*

Hence, to check installability of a package u, v in a repository R, we can pass the constraint set R_c to any CP solver and ask whether $X_u = v$ is satisfiable. We can also simply ask whether there exist a version of a unit that is installable, by asking whether $X_u > 0$ is satisfiable.

Chapter 7

Package management meta-tools: survey and state of the art

In this chapter, we will consider existing package management tools. These tools perform functions such as storage management, distribution and dependency management.

- *Storage management* includes packing and unpacking packages, checking their integrity, setting up configuration files.
- *Distribution* is the network transfer of package files.
- *Dependency management* orchestrates the previous two tasks in order to install, remove or upgrade software packages while maintaining the health of the system.

Major Linux distributions tend to use different tools for different tasks. Often a low-level storage management tool such as `dpkg` for Debian-based systems or `rpm` for RPM-based systems is controlled by a higher-level "meta" tool such as APT or urpmi. Ports-based systems such as Gentoo may use a single tool (emerge).

We are mostly interested in the formal aspects of dependency modeling and in the algorithmic aspects of dependency solving. Thus, low-level tools are out of our scope, but it is generally difficult to distinguish these two different kinds of tools without a close examination. We therefore begin by doing a quick survey of known tools, open-source and commercial, that deal with package management – some of which are programming language-specific.

Most of the many open-source and commercial package management projects deal only with storage management or distribution. Most of the rest do not handle complex requirements (namely, disjunctive dependencies with conflicts). Of the handful of package management tools that have non-trivial dependency handling logic we have the well-known and mainstream `apt`, `urpmi` and `smart` tools, on which we will focus in the remaining sections, providing for each of these a description of their algorithms, particularities and limitations we found during our

investigations. In particular, we report, for each dependency solver, our findings concerning completeness, good formal properties (commutativity, declarativity), and efficiency. We also tested whether, given a full solution, each depsolver is able to simply verify that it is indeed a solution, and apply it directly.

We also describe some properties of the code (programming language used, number of lines of code, overall structure, code quality).

Finally, we argue that neither of these tools can be used to correctly and efficiently check the abundance and peace conditions for a repository.

7.1 Quick survey of known tools and formalizations

7.1.1 Software providing NP-complete dependency management logic

These are package management tools that handle a dependency system whose expressive power is at least equivalent to boolean logic and that have solvers which should, bugs notwithstanding, be able to solve generic satisfiability problems.

The three major pieces of such a software (namely APT, URPMI and Smart), that are often used by well-established Linux distributions, are studied extensively in the following section. For completeness, we mention the following minor managers that also have a comparable level of logic complexity:

- `ipkg`, also known as `The Itsy package management system`. Although it is proclaimed as a lightweight `dpkg`-replacement for iPaq PDAs, it has a complete dependency resolution engine. It is written in C.

- `swup` A high-level package management tool, written in Python and used in Trustix Linux, that is able to handle conflicts, disjunctive dependencies and to install multiple versions (in fact, Gentoo-like slots) of a given unit. This tool may have been taking a non-OSS path as its CVS repository and source code seem to be no longer available.

- `Mongoose Package Manager`, also known as `mpak`, was aiming to be "a kernel and architecture-independent package manager with support for dependency tracking...". Written in C++, it is largely unfinished and was last updated in August 2004.

- `slapt-get` An APT-like system for Slackware package management written in C which handles disjunctive dependencies, conflicts and suggestions, but does not seem to do complete resolution.

Derivatives of the major package management tools:

- `fink` is a port of the Debian APT system to Darwin and Mac OS X.

7.1.2 Entities handling less-than-NP-complete dependency logic

Installation-on-demand

These tools attempt to provide automatic, on-demand and transparent installation of software.

- `klik` This tool is described more in detail in Chapter 4.
- `ZeroInstall` http://0install.net/ It is composed by two systems, Filesystem and Injector allowing users to run software without needing neither installation nor superuser privileges.

Storage managers

The tools in this category do not compile packages. Their main task is to add and remove packages. To add a package, the user obtains a package file (a binary archive containing some metadata) manually and then invokes the tool. The tool then checks the integrity of the package, unpacks it, calls contained installation scripts, and registers the package in the system package database. To remove a package, the user simply invokes the tool with the name of the package to remove. The associated files, whose paths are stored in the package database, are removed, unless they are configuration files modified by the user.

Classical storage-only managers The following storage-only managers do not know anything about dependencies or conflicts.

- FreeBSD, NetBSD and OpenBSD `pkg_add, pkg_remove` tools that complement the ports system.
- Slackware's `pkgtool` similar to the above BSD tools.
- `UniPKG` A modular package manager that natively handles various package formats.
- `gnupdate` A universal package management system comprising `gpkg` and `libpackman`.
- `libpackage` (Open Package Library)
- `Splack` A port of Slackware to SPARC.
- `UPMS` is a simple package management system written in shell script with simple dependency management, for Filesystem Hierarchy Standard (FHS)-compliant Linux distributions.
- `pkgutils` are package management utilities for Linux, used by the CRUX distribution. They are written in C++ and have no dependency management, only storage management with file-based conflict detection.

- `tinypackage` A lightweight package manager for UNIX systems.

Some storage managers know of dependencies and conflicts because they are actually the backend of a suite of higher-level package management tools

- `dpkg` as used by Debian APT
- `rpm` as used either directly or as `librpm` by RPM-based distributions such as Red Hat or Madriva.
- `xpkg` is the package manager of the OpenDarwin project handles complex dependencies.

Symbolic link managers A particular and most primitive class of storage managers is the class of symbolic link managers. Assume software packages come as simple `tar` archives that can be extracted anywhere. Extending all search paths (for executables, libraries, manual pages...) for each piece of new software is cumbersome (one would get very long path variables) requires users to restart their session to take fully effect. As for extracting an archive directly into the root filesystem, not only is it very dangerous (as vital system files, such as `/etc/passwd` may get, erroneously or maliciously, overwritten), it is also impossible to undo. Indeed, even if no file is overwritten, one still needs the original archive to know which files to delete. The idea of symbolic link managers is to have each piece of software in its own directory, at a known place, for instance under `/opt` or `/usr/depot`, to confine software to their own directories, thus allowing easy removal and preventing modification of important system files. Links are then established from standard points in the filesystem to that directory. For instance, `vim` would be unpacked in `/opt/vim/`, and a symbolic link would be created by the manager from `/usr/bin/vim` to `/opt/vim/bin/vim`. Removing packages is as easy as removing the `/opt/vim` directory and its contents while removing associated symbolic links. This scheme has the advantage of extreme simplicity. On modern systems, the overhead of symbolic links should be negligible. The `depot` tool is the most well-known symbolic link manager which has inspired a number of others.

- `depot`
- `GNU stow` and derivatives like `stowES`, `XStow`, `reflect`
- `graft`
- `swpkg`
- `encap`, `epkg`, `sencap`
- `slashpackage`
- `spill`
- It package manager

Classical storage and download managers

Augmenting storage managers with simple (non-disjunctive, non-conflictual) dependency information and automatic retrieval capabilities gives a class of tools for which the linear-time topological sorting algorithm is adequate for solving dependency problems. Unfortunately, disjunctions and conflicts can only be disposed of in an ideal world.

- `aduva`
- `uludag`
- `spkg` A package management utility for Slackware that can do upgrades.
- `openbechede` An OpenBSD packages tool

Package managers for programming languages

Modern programming languages have built-in modularity and therefore are ideally suited for software component management techniques. However this also means that conflicts and disjunctions are not common, and component managers seldom handle them.

- `Godi` for Ocaml features dependency and conflict resolution, compilation options and automated downloading and compiling.
- Ruby gems
- `libneedle` for Ruby
- `CPAN`, the *Comprehensive Perl Archive Network,* contains package installation tools that try to automate the download and installation of `Perl` modules
- *OSGi bundles are Java archive files that contain some installation and configuration classes and some static dependency information.*

Recompilation frameworks

In their basic implementation, these systems can be described as a structured version control repository. Units exist as subdirectories of the repository. Each subdirectory contains a Makefile and associated scripts. Users of the system will first *checkout* the latest version of the repository. They will then place themselves in the directory of a unit they wish to install. An invocation of `make` will then automatically download the sources, configure and compile them, and install the unit. If the unit depends on other units, a recursive invocation of make on the associated directory will download and build it. The typical representative of this class of systems the BSD ports system, as used by FreeBSD, NetBSD and OpenBSD. These use standard tools (CVS, make, fetch) and shell scripts. While simple and

elegant, the repositories being very large, performance issues arise when doing updates with CVS or Subversion (SVN). Also, the lack of logical expressiveness of `make` prohibits circular or disjunctive dependencies and conflicts.

Gentoo Linux significantly improves the ports system by replacing make with a more intelligent tool called `emerge`. This tool has finer-grained control over compilation options and can handle disjunctive dependencies. The general architecture and the "slotting" system reduce the number of conflicts. However, conflicts can arise due to incompatible requested version ranges, especially if subdistributions of different stability are mixed in the same system. Conflicts and circular dependencies, although rare, are not well-handled by `emerge`.

Version control system-based:

- BSD ports, the classical CVS and make-based ports system
- Gentoo portage is basically BSD ports with enhanced logic
- `mpkg` A FreeBSD-like ports collection for DEC OSF/1, Linux, Solaris.
- Darwin `pkg` is a port of the FreeBSD software package system to the Darwin OS.
- The `Scrudgeware` tools are a combination of Encap and of the BSD ports system.
- `Emerde` is a port of Gentoo's Portage system for other distributions.
- `Conary` is an interesting package management system that is based on an enhanced source version control system including local changesets (describing local modifications to configuration files) and shadows (which allows creation of a branch in the source control system that follows the evolutions of its parents). It is used in rPath linux.

Recompilation-only:

- `toast` is a simple source-and-symlinks, recompiling package manager for both root and non-root users.

Testing tools

`Piuparts` (acronym for **p**ackage **i**nstallation, **up**grading, and **r**emoval **t**esting **s**uite) is a tool to test the installation and upgrade of a debian package relative to a given debian distribution. Furthermore, it checks that removal of a package using the standard debian mechanism leaves no files behind. More precisely, the tool permits to perform three kinds of tests:

1. test whether a package can be properly installed and removed relative to a given distribution,

2. test whether a package can be upgraded to a specific new version relative to a given distribution,

3. test whether a first given distribution can be upgraded to a second given distribution.

The tests of installation and upgrade are performed using the debian standard commands `apt-get install` in case of testing installation or upgrade of a single package, and `apt-get dist-upgrade` in case of testing upgrade of a distribution. These installation operations are done in a fresh debian installation of the chosen base distribution. An important point here is that all this tool does is *testing*, no formal verification is done. As a consequence, this tool is subject to the same limitations as `apt` itself, but this is deliberate since the purpose of this tool is exactly to test installation and upgrade using the standard debian tools.

Metadata formats or ontologies

Modular programming is not something new and formalization of interdependencies between software components is as old as dependency graphs and the `make` tool. Of course such crude modeling is grossly insufficient for today's highly complex relationships between packages. Heterogeneous software environments mandate, because of binary compatibility and linking issues, the use of version numbers, dependencies with upper- and lower-bounded ranges, disjunctive dependencies and therefore conflicts. There is a limit on what can be done with dynamic linking, hence many important aspects of a software product, such as the graphical toolkit it uses or the optimized numerical algorithm library it relies on must be known as a compile time. As a result, the number of units a software product can give grows as the product of the number of variants of components that must be statically linked. Assume we have a video player with GTK, KDE or command-line interfaces whose codecs must for some reason be statically linked (perhaps due to the language used), with available unoptimized codecs, optimized for AMD and Intel, 32- and 64-bit versions. This gives a total of $3 \times 2 \times 2 = 12$ units. Add two units for common architecture-independent files such as documentation and GUI skin files. This gives 14 units. On the other hand, if we make the product into a library with its own documentation and development version, then we have 4 library units, 2 development units, 2 architecture-independant units and 3 main units with different interfaces, which saves us one unit.

- *Trove* is a somewhat old proposal by Eric S. Raymond focused on software archive maintenance issues. It proposes to use the same dependency fields as in Debian, but does not delve deeper into the associated combinatorial problems.

- *The Dublin Core Metadata Initiative* is a large and ambitious initiative that aims to create interoperable metadata standards for describing all kinds of

information resources. Its overly broad scope and fuzzy semantics are not suitable for package management.

- *RPM metadata* is an XML format proposal by Duke University for describing package metadata. From a dependency viewpoint, their DTD is based on the RPM format and provides sufficiently expressive fields.
- The *Installable Unit Package Format Specification* is a W3C proposal developed by the OASIS consortium for standardizing packaging formats, notably for commercial software to be distributed on removable media, in order to create interoperable installers, notably for operating systems that lack packaging functions, such as Microsoft Windows, as can be deduced from the participation of InstallShield Softwarea Corp.
- Not to be confused with Fink, *Flink* aims for formalize various knowledge about the Linux system. This includes formalizing package interrelationships. They provide an Owl ontology modeled on the Debian policy manual.

7.2 Analysis of some package management tools

In the following sections we perform and in-depth analysis and description of four mainstream package management meta-tools: Apt, Portage, Smart and Urpmi.

We are particularly interested in determining their fitness for being used as installability verifiers, i.e. tools to check server-side repository consistency, which is one of the main goals of our workpackage.

7.2.1 General analysis on a given testbench

For each of these tools, we performed the very same tests on a specially built package base, shown in figure 7.1, designed in order to verify completeness of the dependency solving algorithm (crucial for *our* goals), but also the quality of the solution found, which is crucial for the stated goals of these tools, which is maintaining an up-to-date installation on a user machine.

This fake repository contains a deeply nested conflict among glass version 2 and tyre version 2, and has one “optimum” solution (figure 7.2), in terms of number of freshest installed packages, which is given by taking tyre=1, and the most recent version for all other packages.

For information, the WP2 toolchain statistics on this fake repository are available online at the following URL: http://www.edos-project.org/xwiki/stats/car.html. See also figure 7.3.

The analysis show that no system is able to find the best solution but urpmi under certain conditions, and all but Smart do fail in being complete.

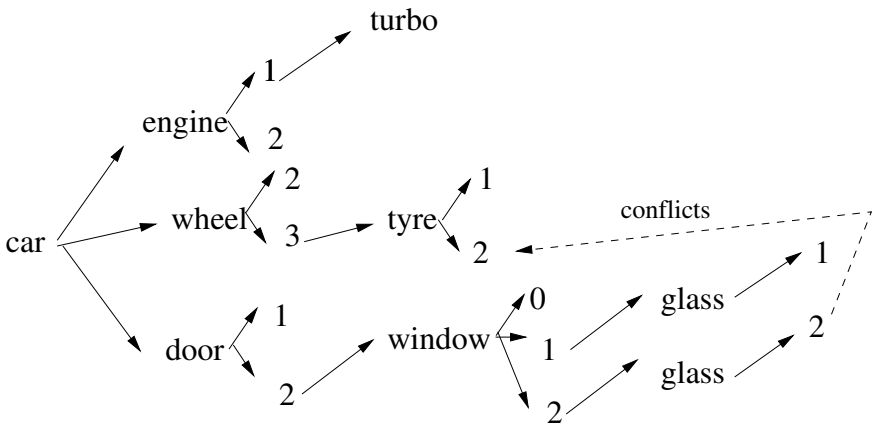


Figure 7.1: Graph of car dependencies and conflicts.

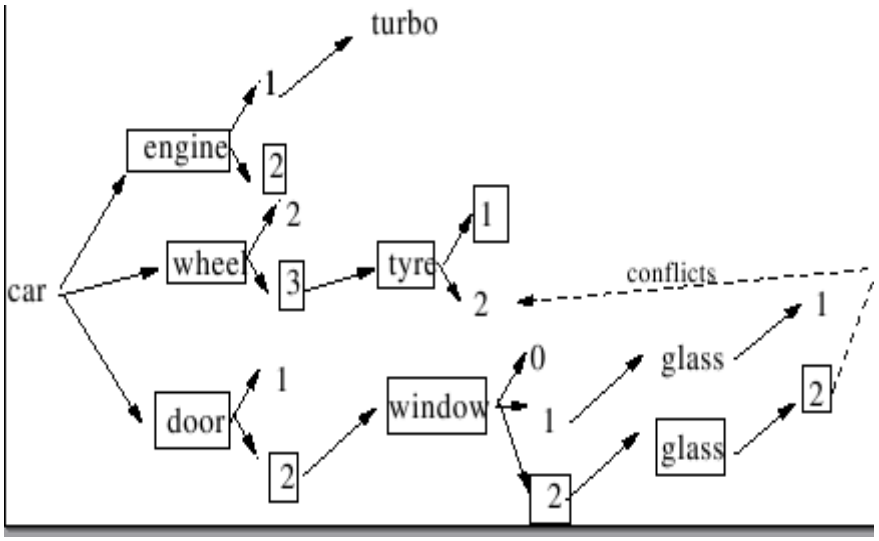


Figure 7.2: Car/Glass: optimum solution (maximum number of latest versions).

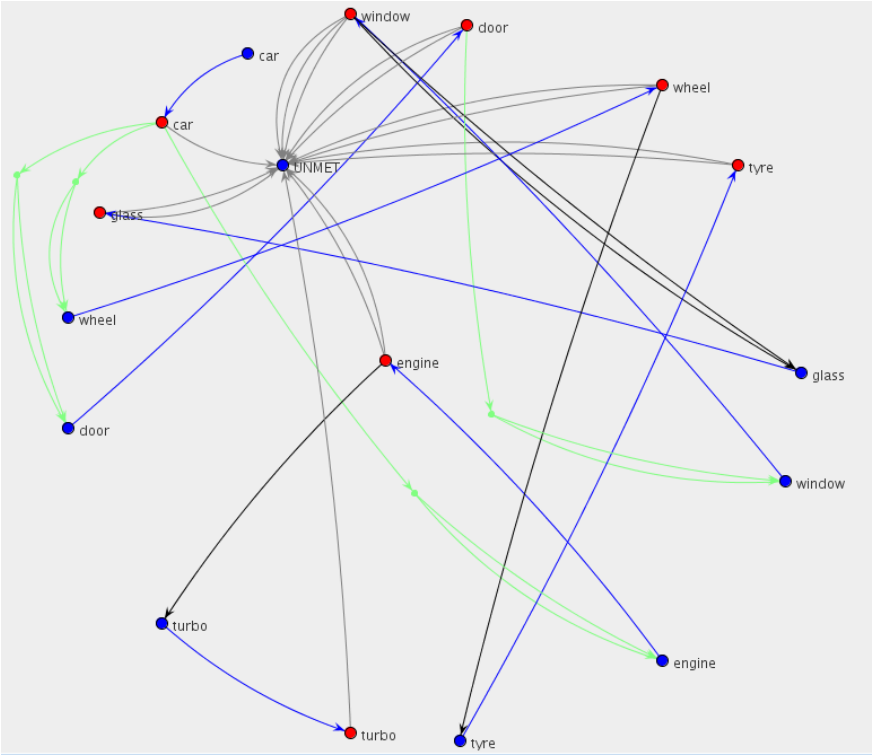


Figure 7.3: Car/Glass: graphical representation of the dependency generated with the EDOS toolchain

7.2.2 Further investigation

For the Apt and Smart tool, we also investigate much more in depth the inner working of the system, finding a very surprising behavior for Apt, and exposing the potentially explosive computational behavior of Smart.

For all these reasons, we conclude that not a single one of these tools can be used to perform installability checking, and that we need to develop our own.

7.3 APT

The APT tool has been for a long time a key element of the success of the Debian distribution. It was indeed one of the first meta-package management tools incorporating dependency solving and package retrieving algorithms that gave the user the feel of a system able to automatically fetch and install the best set of packages suited for her needs.

The problem that APT tries to solve is quite tricky: maintain a distribution consistent, while upgrading to the most recent version of the packages that a user may require, which, as we have said before, is a more complex problem than mere installability, known now to be NP-complete.

As a result, in order to get answers in acceptable time, APT is forced to incorporate heuristics and stategies that turn out, when properly analysed, to be incomplete, inconsistent and to exhibit a surprising behavior (for a user).

For these reasons, APT is not an acceptable tool if one only needs to check for installability of a package (and it should not be used in the Debian production process to check whether a package may or not be migrated from one release to another, like from `unstable` to `testing`).

We report here our findings, that seem not widely known.
Let's start by testing APT's dependency solver on the Car/Glass example.

7.3.1 Apt on the Car/Glass testbench

If `apt-get` was optimal, it should install the marked packages in the figure 7.2.
Or, an acceptable alternative would be the packages marked in figure 7.4.
However, apt-get doesn't find any of these solutions[1]:

[1]The tests reported here were performed on CaixaMagica premises using apt for rpm, which uses the very same dependency solver as apt, so the results are transferable to apt in general, but w.r.t. rpm repositories.

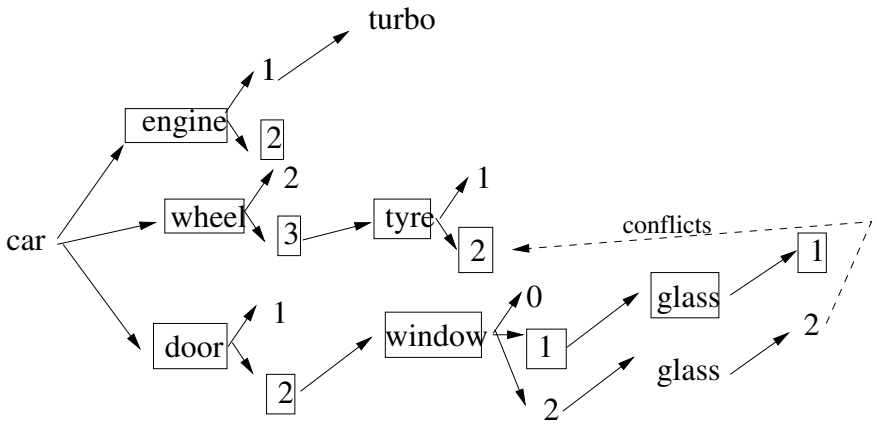


Figure 7.4: Car/Glass: suboptimal solution

```
sclara:/etc/apt/sources.list.d # apt-get install car
Reading Package Lists... Done
Building Dependency Tree... Done
[...]

The following packages have unmet dependencies:
  car: Depends: wheel (>= 2) but it is not going to be installed
E: Broken packages
```

This comes from Apt's choice of always installing the greatest version of a package, that lead it to the following steps

1. Installed "engine = 2"
2. Installed "wheel = 3" and for that installed its dependency "tyre = 2"
3. Tried to install "door", but before tried to install its dependency "window = 2" but even before tried to install window dependency's "glass = 2". Since this last one failed, they all failed.

Quite evidently, Apt does not backtrack to look for one of the two possible solutions.

Scenario 1 - fix "wheel = 3"

Let's investigate more why apt does not backtrack: we manually install wheel and then door, in this order:

1. Installed wheel (and apt-get installed well version 3 and tyre = 2)
2. Tried to install door but apt-get suggest to remove wheel and tyre, before install door + windows + glass (see next output)

The install operation of “door.rpm” asks permission for removing the previous installed packages:

```
  sclara:/etc/apt/sources.list.d # apt-get install door
 [...]
The following packages will be REMOVED:
  tyre wheel
The following NEW packages will be installed:
  door glass window
```

Why apt-get had not tried to install window = 1 and glass = 1?

Scenario 2 - fix “window = 1”

Notice that the installation of the package “window = 1” alone succeds (and it installs also its dependency “glass = 1”), and then we can install door without problems:

```
  sclara:/etc/apt/sources.list.d # apt-get install door
[..,]
Committing changes...
Preparing...                ########################################### [100%]
   1:door                   ########################################### [100%]
Done.
```

Scenario 3 - fix “wheel = 2”

Let’s explore the other possibility: fix “wheel = 2” by directly installing wheel = 2. This operation succeeds and then we can install “car.rpm” and its dependencies:

```
  Reading Package Lists... Done
Building Dependency Tree... Done
[...]
Do you want to continue? [Y/n] Y
Committing changes...
Preparing...                ########################################### [100%]
   1:engine                 ########################################### [ 20%]
   2:glass                  ########################################### [ 40%]
   3:window                 ########################################### [ 60%]
   4:door                   ########################################### [ 80%]
   5:car                    ########################################### [100%]
Done.
```

7.3.2 Algorithm specification

The previous tests highlight the following behavior of the Apt:

1. Check the dependencies of a package
2. Try to install dependencies one-by-one in the order they are presented in the RPM.
3. For each dependency, try to install its sub-dependencies by the greater version presented.
4. If one subdependency fail by conflict with a package that will be installed (tyre=2 conflicted with glass=2), then the install operation aborts. It does not try to backtrack and check lesser versions (tyre = 1, for instance, or even window = 1).
5. If one subdependency fail by conflict with a package that is already installed (case where wheel and tyre were installed), then the install prompts for removal of the installed package. It does not try to see if there is alternatives for the conflict package.

This heuristic tends to work well with well-behaviored repositories such as the ones centralized by a commercial distribution. It sacrifies the quality of the solution for the speed of the analysis.

7.3.3 Apt's surprising behavior.

We have also tested Apt's behavior on a snapshot of the Debian pool taken in the middle of 2005, and available in the EDOS subversion repository as `Data/Sources/Packages-pool.gz`. Of the many tests performed, we retain the following three, which clearly exhibit some of Apt's limitations.

- `apt-get install abiword-gnome=2.2.7-3` **fails**
- `apt-get install abiword-gnome=2.2.7-3 abiword-common=2.2.7-3` succeeds
- `apt-get install abiword-common=2.2.7-3 abiword-gnome=2.2.7-3` **succeeds, but installs one more package!**

First test, failure for abiword-gnome=2.2.7-3

```
Running apt using fake directory structure /extended/tmp/apt
Populating ...done.
Creating fake configuration file
Creating fake source list file
Updating debian-pool cache
```

```
Atteint http://www.pps.jussieu.fr unstable/main Packages
Ign http://www.pps.jussieu.fr unstable/main Release
Lecture des listes de paquets...
Trying to install abiword-gnome=2.2.7-3
Lecture des listes de paquets...
Construction de l'arbre des dépendances...
Certains paquets ne peuvent être installés. Ceci peut signifier
que vous avez demandé l'impossible, ou bien, si vous utilisez
la distribution unstable, que certains paquets n'ont pas encore
été créés ou ne sont pas sortis d'Incoming.

Puisque vous n'avez demandé qu'une seule opération, le paquet n'est
probablement pas installable et vous devriez envoyer un rapport de bogue.
L'information suivante devrait vous aider à résoudre la situation:

Les paquets suivants contiennent des dépendances non satisfaites:
  abiword-gnome: Dépend: abiword-common (= 2.2.7-3) mais 2.2.9-1
  devra être installé
```

Second test, success for abiword-gnome=2.2.7-3 abiword-common=2.2.7-3

```
Running apt using fake directory structure /extended/tmp/apt
Populating ...done.
Creating fake configuration file
Creating fake source list file
Updating debian-pool cache
Atteint http://www.pps.jussieu.fr unstable/main Packages
Ign http://www.pps.jussieu.fr unstable/main Release
Reading Package Lists... Done
Trying to install abiword-gnome=2.2.7-3 abiword-common=2.2.7-3
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  abiword-common abiword-gnome adduser coreutils cpp cpp-4.0 dbus-1
  dbus-glib-1 debconf debconf-i18n debianutils defoma esound-common file
  fontconfig gcc-3.3-base gcc-4.0-base gconf2 gnome-keyring gnome-mime-data
  gsfonts libacl1 libart-2.0-2 libaspell15 libatk1.0-0 libattr1 libaudiofile0
  libbonobo2-0 libbonobo2-common libbonoboui2-0 libbonoboui2-common libbz2-1.0
  libc6 libcap1 libcdparanoia0 libcomerr2 libcupsys2-gnutls10 libdb3
  libdb4.2 libenchant1 libesd0 libexpat1 libfam0c102 libfontconfig1
  libfreetype6 libfribidi0 libgcc1 libgconf2-4 libgcrypt11 libgdbm3
  libglade2-0 libglib2.0-0 libgnome-keyring0 libgnome2-0
  libgnome2-common libgnomecanvas2-0 libgnomecanvas2-common
  libgnomecups1.0-1 libgnomeprint2.2-0 libgnomeprint2.2-data
  libgnomeprintui2.2-0 libgnomeprintui2.2-common libgtk2.0-common
  libgnomeui-0 libgnomeui-common libgnomevfs2-0 libgnomevfs2-common
  libgnutls11 libgpg-error0 libgtk2.0-0 libgtk2.0-bin libmyspell3
  libgucharmap4 libhal-storage0 libhal0 libice6 libidl0 libjpeg62
```

```
  libkrb53 libldap2 liblocale-gettext-perl liblzo1 libmagic1
  libncurses5 libnewt0.51 libopencdk8 liborbit2 libpam-modules
  libpam-runtime libpam0g libpango1.0-0 libpango1.0-common
  libperl5.8 libpng12-0 libpopt0 libsasl2 libslang2 libsm6
  libsmbclient libstdc++5 libstdc++6 libtasn1-2
  libtext-charwidth-perl libtext-iconv-perl libtext-wrapi18n-perl
  libtiff4 libx11-6 libxcursor1 libxext6 libxft2 libxi6 libxinerama1
  libxml2 libxrandr2 libxrender1 login lsb-base ncurses-bin
  passwd perl perl-base perl-modules sed shared-mime-info
  ttf-bitstream-vera ucf whiptail x11-common xlibs-data zlib1g

Suggested packages:
  abiword-plugins abiword-plugins-gnome abiword-doc xfs cpp-doc
  cpp-2.95-doc gcc-4.0-locales debconf-doc debconf-utils
  libterm-readline-gnu-perl libgnome2-perl libqt-perl
  libnet-ldap-perl libgnome-perl defoma-doc psfontmgr
  dfontmgr aspell aspell-bin libbz2-dev bzip2 glibc-doc esound
  libfreetype6-dev rng-tools gnome-icon-theme gnutls-bin krb5-doc
  krb5-user libpam-doc ttf-kochi-gothic ttf-kochi-mincho
  ttf-thryomanes ttf-baekmuk ttf-arphic-gbsn00lp ttf-arphic-bsmi00lp
  ttf-arphic-gkai00mp ttf-arphic-bkai00mp libterm-readline-perl-perl
  x-window-system-core x-window-system

Recommended packages:
  abiword-help aspell-en aspell6-dictionary abiword-gtk xfonts-abi
  x-ttcidfont-conf apt-utils libft-perl libatk1.0-data
  esound-clients python-xmlbase libglib2.0-data gamin
  hicolor-icon-theme myspell-en-us myspell-dictionary libgpmg1
  libsasl2-modules xml-core perl-doc fam

The following NEW packages will be installed:
  abiword-common abiword-gnome adduser coreutils cpp cpp-4.0 dbus-1
  dbus-glib-1 debconf debconf-i18n debianutils defoma esound-common
  file fontconfig gcc-3.3-base gcc-4.0-base gconf2 gnome-keyring
  gnome-mime-data gsfonts libacl1 libart-2.0-2 libaspell15 libatk1.0-0
  libattr1 libaudiofile0 libbonobo2-0 libbonobo2-common libbonoboui2-0
  libbonoboui2-common libbz2-1.0 libc6 libcap1 libcdparanoia0
  libcupsys2-gnutls10 libdb3 libdb4.2 libenchant1 libesd0 libexpat1
  libfam0c102 libfontconfig1 libfreetype6 libfribidi0 libgcc1
  libgconf2-4 libgcrypt11 libgdbm3 libglade2-0 libglib2.0-0 libcomerr2
  libgnome-keyring0 libgnome2-0 libgnome2-common libgnomecanvas2-0
  libgnomecanvas2-common libgnomecups1.0-1 libgnomeprint2.2-0
  libgnomeprint2.2-data libgnomeprintui2.2-0 libgnomeprintui2.2-common
  libgnomeui-0 libgnomeui-common libgnomevfs2-0 libgnomevfs2-common
  libgnutls11 libgpg-error0 libgtk2.0-0 libgtk2.0-bin libgtk2.0-common
  libgucharmap4 libhal-storage0 libhal0 libice6 libidl0 libjpeg62
  libkrb53 libldap2 liblocale-gettext-perl liblzo1 libmagic1 libmyspell3
  libncurses5 libnewt0.51 libopencdk8 liborbit2 libpam-modules
  libpam-runtime libpam0g libpango1.0-0 libpango1.0-common libperl5.8
```

```
  libpng12-0 libpopt0 libsasl2 libslang2 libsm6 libsmbclient libstdc++5
  libstdc++6 libtasn1-2 libtext-charwidth-perl libtext-iconv-perl
  libtext-wrapi18n-perl libtiff4 libx11-6 libxcursor1
  libxext6 libxft2 libxi6 libxinerama1 libxml2 libxrandr2 libxrender1 login
  lsb-base ncurses-bin passwd perl perl-base perl-modules sed shared-mime-info
  ttf-bitstream-vera ucf whiptail x11-common xlibs-data zlib1g
```
0 packages upgraded, 130 newly installed, 0 to remove and 0 not upgraded.

Third test, different success for abiword-common=2.2.7-3 abiword-gnome=2.2.7-3

In this case, Apt will install also `libaspell15c2`, which is not proposed in the previous example, despite the fact that the commands given by the users differ in the ordering of the arguments.

```
Running apt using fake directory structure /extended/tmp/apt
Populating ...done.
Creating fake configuration file
Creating fake source list file
Updating debian-pool cache
Atteint http://www.pps.jussieu.fr unstable/main Packages
Ign http://www.pps.jussieu.fr unstable/main Release
Reading Package Lists... Done
Trying to install abiword-common=2.2.7-3 abiword-gnome=2.2.7-3
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  abiword-common abiword-gnome adduser coreutils cpp cpp-4.0 dbus-1 dbus-glib-1
  debconf debconf-i18n debianutils defoma esound-common file fontconfig
  gcc-3.3-base gcc-4.0-base gconf2 gnome-keyring gnome-mime-data gsfonts
  libacl1 libart-2.0-2 libaspell15 libaspell15c2 libatk1.0-0 libattr1
  libaudiofile0 libbonobo2-0 libbonobo2-common libbonoboui2-0
  libbonoboui2-common libbz2-1.0 libc6 libcap1 libcdparanoia0 libcomerr2
  libcupsys2-gnutls10 libdb3 libdb4.2 libenchant1 libesd0 libexpat1 libfam0c102
  libfontconfig1 libfreetype6 libfribidi0 libgcc1 libgconf2-4 libgcrypt11
  libgdbm3 libglade2-0 libglib2.0-0 libgnome-keyring0 libgnome2-0
  libgnome2-common libgnomecanvas2-0 libgnomecanvas2-common libgnomecups1.0-1
  libgnomeprint2.2-0 libgnomeprint2.2-data libgnomeprintui2.2-0
  libgnomeprintui2.2-common libgnomeui-0 libgnomeui-common libgnomevfs2-0
  libgnomevfs2-common libgnutls11 libgpg-error0 libgtk2.0-0
  libgtk2.0-bin libgtk2.0-common libgucharmap4 libhal-storage0 libhal0
  libice6 libidl0 libjpeg62 libkrb53 libldap2 liblocale-gettext-perl
  liblzo1 libmagic1 libmyspell3 libncurses5 libnewt0.51 libopencdk8
  liborbit2 libpam-modules libpam-runtime libpam0g libpango1.0-0
  libpango1.0-common libperl5.8 libpng12-0 libpopt0 libsasl2 libslang2
  libsm6 libsmbclient libstdc++5 libstdc++6 libtasn1-2
  libtext-charwidth-perl libtext-iconv-perl libtext-wrapi18n-perl
  libtiff4 libx11-6 libxcursor1 libxext6 libxft2 libxi6
  libxinerama1 libxml2 libxrandr2 libxrender1 login lsb-base
```

```
  ncurses-bin passwd perl perl-base perl-modules sed shared-mime-info
  ttf-bitstream-vera ucf whiptail x11-common xlibs-data zlib1g

Suggested packages:
  abiword-plugins abiword-plugins-gnome abiword-doc xfs cpp-doc
  cpp-2.95-doc gcc-4.0-locales debconf-doc debconf-utils
  libterm-readline-gnu-perl libgnome2-perl libqt-perl libnet-ldap-perl
  libgnome-perl defoma-doc psfontmgr dfontmgr aspell aspell-bin
  libbz2-dev bzip2 glibc-doc esound libfreetype6-dev rng-tools
  gnome-icon-theme gnutls-bin krb5-doc krb5-user libpam-doc
  ttf-kochi-gothic ttf-kochi-mincho ttf-thryomanes ttf-baekmuk
  ttf-arphic-gbsn00lp ttf-arphic-bsmi00lp ttf-arphic-gkai00mp
  ttf-arphic-bkai00mp libterm-readline-perl-perl x-window-system-core
  x-window-system

Recommended packages:
  abiword-help aspell-en aspell6-dictionary abiword-gtk xfonts-abi fam
  x-ttcidfont-conf apt-utils libft-perl libatk1.0-data esound-clients
  python-xmlbase libglib2.0-data gamin hicolor-icon-theme myspell-en-us
  myspell-dictionary libgpmg1 libsasl2-modules xml-core perl-doc

The following NEW packages will be installed:
  abiword-common abiword-gnome adduser coreutils cpp cpp-4.0 dbus-1
  debconf debconf-i18n debianutils defoma esound-common file fontconfig
  gcc-3.3-base gcc-4.0-base gconf2 gnome-keyring gnome-mime-data gsfonts
  libacl1 libart-2.0-2 libaspell15 libaspell15c2 libatk1.0-0 libattr1
  libaudiofile0 libbonobo2-0 libbonobo2-common libbonoboui2-0 dbus-glib-1
  libbonoboui2-common libbz2-1.0 libc6 libcap1 libcdparanoia0 libcomerr2
  libcupsys2-gnutls10 libdb3 libdb4.2 libenchant1 libesd0 libexpat1
  libfontconfig1 libfreetype6 libfribidi0 libgcc1 libgconf2-4 libgcrypt11
  libgdbm3 libglade2-0 libglib2.0-0 libgnome-keyring0 libgnome2-0
  libgnome2-common libgnomecanvas2-0 libgnomecanvas2-common libgnomecups1.0-1
  libgnomeprint2.2-0 libgnomeprint2.2-data libgnomeprintui2.2-0 libfam0c102
  libgnomeprintui2.2-common libgnomeui-0 libgnomeui-common libgnomevfs2-0
  libgnomevfs2-common libgnutls11 libgpg-error0 libgtk2.0-0 libgtk2.0-bin
  libgtk2.0-common libgucharmap4 libhal-storage0 libhal0 libice6 libidl0
  libjpeg62 libkrb53 libldap2 liblocale-gettext-perl liblzo1 libmagic1
  libmyspell3 libncurses5 libnewt0.51 libopencdk8 liborbit2 libpam-modules
  libpam-runtime libpam0g libpango1.0-0 libpango1.0-common libperl5.8
  libpng12-0 libpopt0 libsasl2 libslang2 libsm6 libsmbclient libstdc++5
  libstdc++6 libtasn1-2 libtext-charwidth-perl libtext-iconv-perl
  libtext-wrapi18n-perl libtiff4 libx11-6 libxcursor1 libxext6 libxft2 libxi6
  libxinerama1 libxml2 libxrandr2 libxrender1 login lsb-base ncurses-bin
  passwd perl perl-base perl-modules sed shared-mime-info ttf-bitstream-vera
  ucf whiptail x11-common xlibs-data zlib1g
0 packages upgraded, 131 newly installed, 0 to remove and 0  not upgraded.
```

7.3.4 Conclusions on APT

It is quite clear now that

Apt is not complete : the first test shows that it does not find a solution for installing `abiword-gnome=2.2.7-3`, while there are many such solutions. This is enough to rule out it as a candidate tool for checking installability.

Apt solutions are order-dependent : the second and third test only differ in the order of the parameters, not their value, yet the solutions found are different. This show that the solution set is order dependent, which is not stated in the documentaiton, and is quite surprising for a user.
Nevertheless, this is perfectly consistent with the fact that the depsolver in APT examines dependencies in a left-to-right order.
This is actually used *on purpose* by many packagers to specify dependencies in a preferred order, like in cases where one finds

```
apache|tomcat5|httpd
```

which is totally silly in a declarative world, as apache and tomcat5 both provide httpd, but makes perfect sense if order matters, as the maintainer is saying that she prefer apache over tomcat5 and tomcat5 over all other httpd servers.

7.3.5 A sidenote: upgradeability in practice, and a suggestion for the future

Another point we want to stress is the extreme user-unfriendliness of the APT tool in the rare occasions when upgrading or installing one package ends up into a major overhaul of the user installation... Here follows a real-world example recorded by one of the authors of this report during his daily running of his beloved Debian-based machine.

```
sudo apt-get install debhelper
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  armagetron armagetron-common autoconf bonobo-activation codebreaker
  debconf debconf-i18n debconf-utils dialog esound-common fb-music-high
  fontconfig frozen-bubble-data grepmail gv intltool-debian
  libaiksaurus-data libaiksaurus0c102 libatk1.0-0 libatk1.0-dev
  libbonobo-activation4 libbonobo2-0 libbonobo2-common libdb3
  libdbd-mysql-perl libdbi-perl libeel2-data libesd0
  libfilehandle-unget-perl libfontconfig1 libforms1 libfreetype6
  libfreetype6-dev libgcc1 libgcrypt1 libgdbm3 libgladexml-perl
  libglib2.0-0 libglib2.0-dev libgnome-perl libgnutls7 libgsf-1
```

```
  libgtk-imlib-perl libgtk-perl libgtk1.2 libgtk1.2-common libgtk1.2-dbg
  libhtml-parser-perl libice-dev libice6 libidl0 liblinc1
  liblocale-gettext-perl liblzo1 libmagick5.5.7
  libmail-mbox-messageparser-perl libmysqlclient12 libncurses5
  libncurses5-dev libncursesw5 libnet-daemon-perl libnet-perl libnewt0.51
  libogg-dev libogg0 liborbit2 libpaper1 libplrpc-perl libpng12-0
  libpopt-dev libpopt0 libsdl-console libsdl-gfx1.2 libsdl-image1.2
  libsdl-ttf1.2 libsdl-ttf2.0-0 libsdl1.2debian libsdl1.2debian-oss
  libsm-dev libsm6 libsmpeg0 libssl0.9.7 libstartup-notification0
  libstdc++5 libt1-5 libtext-charwidth-perl libtext-iconv-perl
  libtext-wrapi18n-perl libtiff-tools libwmf0.2-7 libwww-perl libx11-6
  libx11-dev libxaw7 libxaw7-dev libxcursor1 libxext-dev libxext6
  libxft1 libxft2 libxi-dev libxi6 libxml-parser-perl libxml2 libxmu-dev
  libxmu6 libxmuu-dev libxmuu1 libxp-dev libxp6 libxpm-dev libxpm4
  libxrandr-dev libxrandr2 libxrender-dev libxrender1 libxt-dev libxt6
  libxtrap-dev libxtrap6 libxtst-dev libxtst6 libxv-dev libxv1 lyx
  lyx-common lyx-xforms perl perl-base perl-modules perlmagick pkg-config
  pm-dev po-debconf render-dev tcl8.4 tcl8.4-dev tktable transfig ucf
  whiptail x-dev xaw3dg xbase-clients xfig xfree86-common xlibmesa-dri
  xlibmesa-gl xlibmesa-gl-dev xlibosmesa-dev xlibosmesa4 xlibs xlibs-data
  xpdf-common

The following packages will be REMOVED:
  autoconf2.13 frozen-bubble frozen-bubble-lib gconf2 gnomemeeting
  itk3.1-dev libbonoboui2-0 libbonoboui2-common libdigest-md5-perl
  libforms0.89 libgconf2-4 libgnome2-0 libgnome2-common libgnomeui-0
  libgnomevfs2-0 libgnomevfs2-common libgtk1.2-dev libgtk2.0-0png3
  libgtk2.0-dev libmime-base64-perl libpango1.0-dev libsdl-mixer1.2-dev
  libsdl-perl libsdl-ttf1.2-dev libsdl1.2-dev libsmpeg-dev
  libstorable-perl nautilus tk8.3-dev tktable-dev x-window-system
  x-window-system-core xaw3dg-dev xlib6g xlib6g-dev xlibmesa-dev
  xlibmesa3 xlibosmesa3 xlibs-dev xlibs-pic xpdf xpdf-reader

The following NEW packages will be installed:
  armagetron-common debconf-i18n fb-music-high fontconfig intltool-debian
  libaiksaurus-data libaiksaurus0c102 libeel2-data libfilehandle-unget-perl
  libfontconfig1 libforms1 libgdbm3 libgnutls7 libgsf-1 libice-dev libice6
  libidl0 liblzo1 libmagick5.5.7 libmail-mbox-messageparser-perl
  libmysqlclient12 libncursesw5 libnet-daemon-perl libnewt0.51 libpaper1
  libplrpc-perl libsdl-console libsdl-gfx1.2 libsdl-ttf2.0-0 libsm-dev
  libsm6 libssl0.9.7 libstartup-notification0 libt1-5 libtext-charwidth-perl
  libtext-wrapi18n-perl libtiff-tools libwmf0.2-7 libx11-6 libx11-dev
  libxcursor1 libxext-dev libxext6 libxft1 libxft2 libxi-dev libxi6 libxmu-dev
  libxmu6 libxmuu-dev libxmuu1 libxp-dev libxp6 libxpm-dev libxpm4 libxrandr-dev
  libxrandr2 libxrender-dev libxrender1 libxt-dev libxt6 libxtrap-dev libxtrap6
  libxtst-dev libxtst6 libxv-dev libxv1 lyx-common lyx-xforms pm-dev po-debconf
  render-dev tcl8.4 tcl8.4-dev ucf x-dev xlibmesa-dri xlibmesa-gl
  xlibmesa-gl-dev xlibs-data
75 packages upgraded, 80 newly installed, 42 to remove and 858  not upgraded.
```

```
Need to get 67.1MB of archives. After unpacking 26.9MB will be used.
Do you want to continue? [Y/n] n
```
Abort.

It is quite clear that a careful user is not going to let such an upgade go through unless some hint is given by the system that core functionalities like those suggested by `x-window-system` and `x-window-system-core`, which the tool wants to *remove*, will not disappear, but will be properly replaced by some of the 80 new packages whose installation is suggested.

In other terms, next generation depsolvers will need to *explain* in a reasonable human-readable form the solution they found to the installation or upgrade problem, in order for the user to take an informed action about what they propose to do. This comment is clearly *not* limited to the APT tool, but extends to all tools we have tested insofar.

7.4 Portage

This tool is different from other Linux distributions since it is inspired in BSD ports system. In this last case, we download the source code, unpack in a directory (e.g. /usr/port) and compile it. Portage allow you to update your package tree over the internet with *emerge -u world* command. Or you can download only the packages with *emerge –sync*.

But Portage is also a package building and installation system. We can compile and install and application with the command *emerge package*.

When tested on the Car/Glass package tree[2], Portage fails to install the packages and blocks on the following conflicts:

```
z10n cm-test # emerge -pv --tree  cm-test/car

These are the packages that I would merge, in reverse order:

Calculating dependencies ...done!
[blocks B     ] =cm-test/tyre-2 (is blocking cm-test/glass-2)
[ebuild  N    ] cm-test/car-1  0 kB [1]
[ebuild  N    ]  cm-test/door-2  0 kB [1]
[ebuild  N    ]   cm-test/window-2  0 kB [1]
[ebuild  N    ]    cm-test/glass-2  0 kB [1]
[ebuild  N    ]  cm-test/wheel-3  0 kB [1]
[ebuild  N    ]   cm-test/tyre-2  0 kB [1]
[ebuild  N    ]  cm-test/engine-2  0 kB [1]

Total size of downloads: 0 kB
```

[2] Package build and tests for Portage had been done by Mário Morgado from Caixa Mágica team.

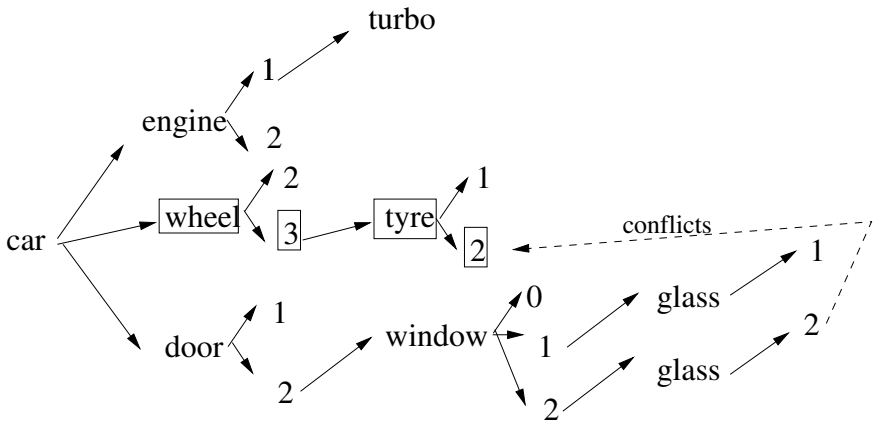


Figure 7.5: Graph of wheel-3 installed

Scenario 1 - fix "wheel = 3"

We can then try to help Protage by installing *wheel-3* before installing the door package.

```
z10n ~ # emerge -pv cm-test/wheel

These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild  N    ] cm-test/tyre-2  0 kB [1]
[ebuild  N    ] cm-test/wheel-3  0 kB [1]
```

The marked packages are now installed (figure 7.5).
If we try to install the door package:

```
z10n ~ # emerge -pv  cm-test/door

These are the packages that I would merge, in order:

Calculating dependencies ...done!
[blocks B     ] =cm-test/tyre-2 (is blocking cm-test/glass-2)
[ebuild  N    ] cm-test/glass-2  0 kB [1]
[ebuild  N    ] cm-test/window-2  0 kB [1]
[ebuild  N    ] cm-test/door-2  0 kB [1]

Total size of downloads: 0 k
```

Portage fails again to find a solution because it does not solve the glass-2 vs tyre-2 conflict.

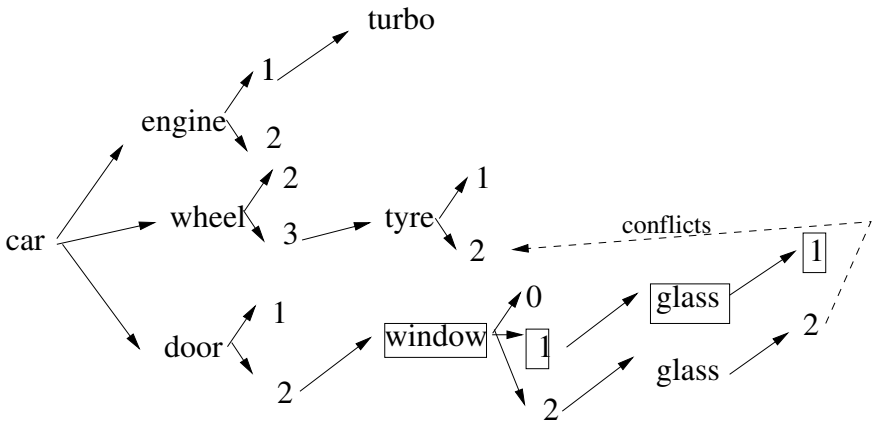


Figure 7.6: Graph of glass-1 and window-1 installed

Scenario 1 - fix "window = 1"

Let's install window-1 before the other packages:

```
  z10n ~ # emerge -pv =cm-test/window-1

These are the packages that I would merge, in order:
Calculating dependencies ...done!
[ebuild  N    ] cm-test/glass-1  0 kB [1]
[ebuild  N    ] cm-test/window-1  0 kB [1]
Total size of downloads: 0 kB
```

Glass-1 and window-1 are installed (figure 7.6).
We can then install door:

```
z10n ~ # emerge -pv cm-test/door
These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild     U ] cm-test/glass-2 [1] 0 kB [1]
[ebuild     U ] cm-test/window-2 [1] 0 kB [1]
[ebuild  N    ] cm-test/door-2  0 kB [1]

Total size of downloads: 0 kB
```

The instalation occurs without problems, and an upgrade of the dependencies (glass and window) is performed. This is different from the *Apt* algorithm behavior.

Installation of car would fail as in the first step since it would conflict with tyre-2.

Scenario 1 - fix “wheel = 2”

This scenario is not worth testing since in last section we discover that all the dependencies are updated as well. The instalation of car will always lead to an update to wheel-3 and the conflict would remain.

7.4.1 Conclusions on Portage

Even without looking at the source code, it is quite clear that the Portage algorithm for finding a solution in an installation problem goes along the following lines:

1. List the dependencies of a package
2. Try to install dependencies one-by-one in the order they are presented in the ebuild package . **If the dependency has an update, update it**.
3. For each dependency, try to install its sub-dependencies by the greater version presented.
4. If one subdependency fail by conflict with a package that will be installed (tyre=2 conflicted with glass=2), then the install operation aborts. It does not try to backtrack and check lesser versions (tyre = 1, for instance, or even window = 1).
5. If one subdependency fail by conflict with a package that is already installed (case where wheel and tyre were installed), then the install exit with a failure message.

It is clear that Portage’s solver is not complete.

7.5 SMART

Smart is a package management system meta tool that offers some advanced feature with respect to package dependency management and installation.

Smart does not depend on any particular package management system but it uses a *plugin* system for defining *backends* that will take care of using a particular package storage management system in order to perform package-related operations. Every *backend* can use one or several *channels* in order to retrieve the needed packages. *Channels* can be of different types (e.g., HTTP, FTP or local repositories) and abstract the differences among the packages retrieval mechanisms.

Smart uses the notion of *transaction* in order to compute a particular set of operations necessary to perform an operation (e.g., installation or upgrade) on a given set of packages. Differently to many other package management meta system that do not make any effort to explore, even partially, the space of the possible solutions

with respect to a given target operation, Smart tries to do so and tries to pick the *"best"* possible solution in that space.

The notion of *"best"* solution is given by a *policy* that weights the found solutions and allows Smart to choose the most suitable one with respect to the chosen policy. There are several predefined *policies* such as "remove the least number of packages" or "upgrade the most number of packages with the less impact", etc. Other policies can be plugged in Smart by extending the relevant classes.

Smart has been proven to been an effective tools that is able to gracefully handle package operations on client installed distributions.

By trying to obtain an *optimal* solution, Smart explores the solution space which is potentially huge, using some heuristics to avoid getting lost in such exploration.

7.5.1 Smart on the Car/Glass testbench

We use again the RPM repository with the Car/Glass packages, and create a Smart in order to perform the tests.

```
Updating cache...                #### [100%]

Fetching information for 'Repositorio Local'...
-> http://localhost/testRPMs/base/release
release                          #### [ 33%]
-> http://localhost/testRPMs/base/release.car
-> http://localhost/testRPMs/base/pkglist.car.bz2
pkglist.car.bz2                  #### [ 66%]
release.car                      #### [100%]
Updating cache...                #### [100%]

Channels have 9 new packages:
    door-1-0@i586
    engine-1-0@i586
    glass-1-0@i586
    turbo-1-0@i586
    tyre-1-0@i586
    tyre-2-0@i586
    wheel-3-0@i586
    window-0-0@i586
    window-1-0@i586
```

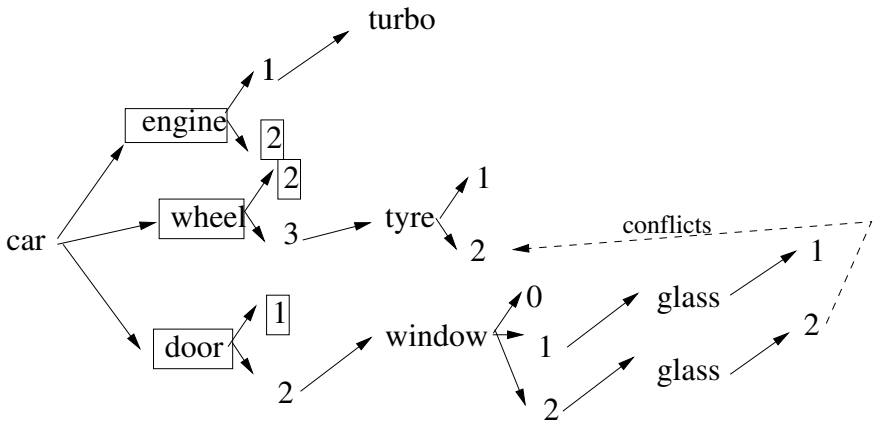


Figure 7.7: Graph of packages installed by Smart for *car*.

```
Saving cache...
```

The result of the command `smart install car` is:

```
  Computing transaction...
Installed packages (4):
  car-2-0@i586      door-1-0@i586     engine-2-0@i586   wheel-2-0@i586
7.8kB of package files are needed.
```

Smart solve the conflicts and installed *car* package in the first attempt. The chosen packages for instalation are a interesting set as we can see in the picture 7.7. Every "branch" that had a conflic with other branch was excluded and a older version of the package was used.

The *wheel-3* package was not installed and *wheel-2* was prefered. The same for *door-1*.

The solution was not optimal because a newer version of wheel (version 3) could be installed without conflict. Or in alternative, "door-2" could be installed.

Let's check some other scenarii.

Scenario 1 - fix "wheel = 3"

In this scenario we will install first the *wheel-3*. This will freeze that branch.

Smart installed the newer versions since no conflict was present:

```
  Computing transaction...
Installed packages (2):
  tyre-2-0@i586     wheel-3-0@i586
```

We then tryed to install the *car* package:

```
Computing transaction...
Installed packages (3):
  car-2-0@i586      door-1-0@i586     engine-2-0@i586
5.8kB of package files are needed.
```

Smart installed the package by guessing well that the only version possible was *door-2*.

Scenario 2 - fix "window = 1"

Now we forced the instalation of window-1 and Smart installed it and its dependency, glass-1.

Now we try to install car. The expected behaviour was the instalation of *wheel-3* since no conflict was expected. Smart had a strange behaviour here and installed wheel-2. It was not necessary.

Scenario 3 - fix "wheel = 2"

Starting by installing wheel-2 Smart could install *door-2* but it had not.

It just installed *door-1* like it had detected that door-2 had a "possible" conflict.

7.5.2 Smart Algorithm

These tests already highlight part of the behevior of Smart's algorithm:

1. check the dependencies of a package
2. try to install dependencies one-by-one
3. for each dependency, try to install its sub-dependencies by the greater version presented. If the greater version has a conflict with a known package (even if the version with the conflict will not be installed) than try to install an older version of the dependency.

Unlike the previous tools, Smarts does try to backtrack and choose older versions of a package when a conflict is found during the state-space exploration. Nevertheless, the backtracking system is not guarantee to find an optimal solution, as we have seem in the previous examples.

7.5.3 Combinatorial explosion

Unfortunately, when performing server-side operations, such as checking the consistency of package bases at the distribution source, the solution space is much bigger than in the average use-case on a client installation, and Smart is unable to

find a solution in an acceptable (or even practically finite) time for a significant number of packages.

We verified this limitation by setting up a package base that comprises the whole Debian Pool, and asking Smart to install a given package starting on an empty system. Here are two examples

23576s = 6h30 to check installability of php3

```
/usr/bin/time smart --data-dir=/ext/tmp/smart install
                    --urls php3
Loading cache...
Updating cache...               ##################### [100%]
Computing transaction...
/pool/main/libg/libgcrypt11/libgcrypt11_1.2.1-4_i386.deb
/pool/main/a/apache/apache-common_1.3.33-7_i386.deb
/pool/main/o/openssl/libssl0.9.7_0.9.7g-1_i386.deb
...
```

23576.80user 6h30 8.71system 13:09:13elapsed 49%CPU

Two months are not enough to check installability of achims-guestbook

```
/usr/bin/time smart --data-dir=/ext/tmp/smart install
                    --urls achims-guestbook
Loading cache...
Updating cache...               ##################### [100%]
Computing transaction...
^C
```

This computation has been stopped two months after being started, without ever returning a solution.

7.5.4 Conclusions on Smart

Smart is up to now the best tool in terms of completeness among the one analysed, even if we have shown that the solutions provided by this tool are not necessarily optimal from a user point of view.

We cannot say anything conclusive about completeness or soundness either: on one side, the depsolver algorithm, using the predefined policies, is only available as Python source code, and not formally described, so we could not really check its correctness; on the other side its explosive behavior has prevented us from being

able to collect experimental evidence of its completeness (or incompleteness).

Nevertheless, the combinatorial explosion exposed by the two tests above is evidently largely enough to rule out Smart as a possible candidate tool for checking installability: our tool must handle dozens of thousands of packages, and cannot spend an unbounded amount of time on checking installability for just one of them.

7.6 URPMI

Urpmi is the depsolver used in the Mandriva distribution; this section contains a description of its inner working as presented by the maintainer of Urpmi himself.

7.6.1 Algorithms used

Basically urpmi constructs a dependency tree from a set of demanded modules. It begins to load the dependency tree for the known set of packages available in its repositories; then a simple tree-walk algorithm is used to gather all required packages. urpmi being an interactive tool, it is able to propose different sets of packages than can solve the set of requirements for the demanded modules. For example, to solve a dependency on ”webfetch” urpmi can use the ”curl” or the ”wget” package, so it will ask the user for it.

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

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

7.6.2 Upgradeability in practice

Given a list of arguments (packages to be installed or upgraded), urpmi produces deterministic results.

urpmi will never downgrade a package. So, if asked to install a package A that requires an older package B than the B currently installed on the system, it will abort.

Some rare situations can make urpmi hang in an infinite loop.

urpme, a counterpart to urpmi, is used to remove packages, and all packages that depend on them recursively.

7.6.3 Notes on implementation

urpmi is written mostly in Perl 5 (about 10,000 lines of code), with a small part in C, used to bind it to the API of the RPM library.

7.6.4 Examples

Here's a simple example of urpmi installing a new package, taking care of conflicts and broken dependencies:

```
sudo urpmi libdb4.2-static-devel
The following packages have to be removed for others
      to be upgraded:
libdb4.1-devel-4.1.25-9mdk.i586
 (due to conflicts with libdb4.2-devel)
libdb4.1-static-devel-4.1.25-9mdk.i586
 (due to unsatisfied db4.1-devel == 4.1.25-9mdk) (y/N) y
To satisfy dependencies, the following packages are going
      to be installed:
libdb4.2-devel-4.2.52-9mdk.i586
libdb4.2-static-devel-4.2.52-9mdk.i586
Proceed with the installation of the 2 packages?
      (43 MB) (Y/n) y
installing libdb4.2-static-devel-4.2.52-9mdk.i586.rpm
           libdb4.2-devel-4.2.52-9mdk.i586.rpm
           from /var/cache/urpmi/rpms
removing libdb4.1-static-devel-4.1.25-9mdk.i586
         libdb4.1-devel-4.1.25-9mdk.i586
Preparing...                    ####################
      1/2: libdb4.2-devel       ####################
      2/2: libdb4.2-static-devel ####################
```

7.6.5 urpmi on the Car/Glass testbench

We use the same RPM repository as before to perform the tests. A naive approach is to tell urpmi to install all the packages in this repository.

Passing all the rpm as arguments

```
# urpmi *.rpm
Some package requested cannot be installed:
door-2-0.i586 (due to missing window-2-0.i586)
engine-1-0.i586
glass-2-0.i586
tyre-1-0.i586
```

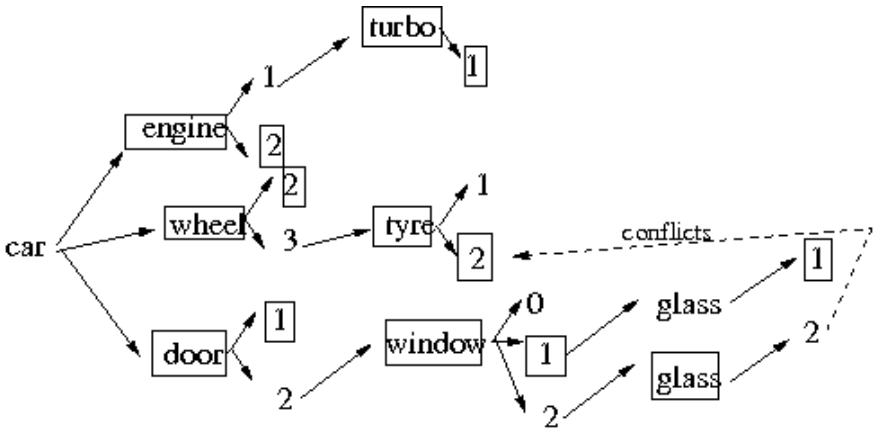


Figure 7.8: Car/Glass: Graph of the packages installed when passed as arguments to urpmi.

```
wheel-3-0.i586
window-0-0.i586
window-2-0.i586 (due to unsatisfied glass[== 2])
Continue? (Y/n)
installing glass-1-0.i586.rpm window-1-0.i586.rpm wheel-2-0.i586.rpm
engine-2-0.i586.rpm tyre-2-0.i586.rpm car-2-0.i586.rpm door-1-0.i586.rpm
turbo-1-0.i586.rpm
Preparing...
      1/8: door                  warning: user prrt does not exist - using root
      2/8: engine                warning: user prrt does not exist - using root
      3/8: wheel                 warning: user prrt does not exist - using root
      4/8: glass                 warning: user prrt does not exist - using root
      5/8: window                warning: user prrt does not exist - using root
      6/8: tyre                  warning: user prrt does not exist - using root
      7/8: car                   warning: user prrt does not exist - using root
      8/8: turbo                 warning: user prrt does not exist - using root
```

The result is basically the same as the one we obtained previously with smart when installing the car from the repository. urpmi solves the conflicts and installs car at the first attempt. However, it does not find the optimum solution. It discarded wheel-3 and also door-2 which are the only 2 branches leading with a potential conflict (figure **??**).

However, this test is far from being representative of a package installation in the real world. To be fair, a RPM repository must be created with a hdlist for urpmi. This is what we are doing in the next section.

Using a repository

First we need to create the repository before we can use it. Then we will try to install car without giving any more clue to urpmi to see what it can find by itself.

```
# genhdlist
# urpmi.addmedia wp2d2-car_test . with hdlist.cz
added medium wp2d2-car_test
wrote config file [/etc/urpmi/urpmi.cfg]
examining synthesis file [/var/lib/urpmi/synthesis.hdlist.The Ultimate Linux
Desktop DVD (Mandriva 2006 Powerpack (local) 1).cz]
copying source hdlist (or synthesis) of "wp2d2-car_test"?
...copying done
/bin/cp: cannot stat ‘/home/EDOS/RPMS.car/pubkey’: No such file or directory
...copying failed
examining hdlist file [/var/cache/urpmi/partial/hdlist.wp2d2-car_test.cz]
writing list file for medium "wp2d2-car_test"
(...)
built hdlist synthesis file for medium "wp2d2-car_test"
found 0 headers in cache
removing 0 obsolete headers in cache
wrote config file [/etc/urpmi/urpmi.cfg]
```

The repository is built and registered with the client (eg. the user laptop) with no particular problem except that urpmi.addmedia complains about a file we did not provide but which is not relevant for this experiment (a pgp key for security checking).

Now, let's try to install the car package.

```
# urpmi car
To satisfy dependencies, the following 7 packages are going to be
installed (0 MB):
car-2-0.i586
door-2-0.i586
engine-2-0.i586
glass-2-0.i586
tyre-2-0.i586
wheel-3-0.i586
window-2-0.i586
Is this OK? (Y/n)
installing tyre-2-0.i586.rpm wheel-3-0.i586.rpm door-2-0.i586.rpm glass-2-0.i58
window-2-0.i586.rpm engine-2-0.i586.rpm car-2-0.i586.rpm from /home/EDOS/RPMS.c
Installation failed:
        tyre = 2 conflicts with glass-2-0.i586
```

Now urpmi is selecting correctly wheel-3 and door-2 but it is failling to install them altogether with the car because it detected the conflict between tyre-2 and glass-2. It appears that urpmi selects always all the freshest versions and it does not backtrack for example to choose tyre-1 instead of tyre-2.

urpmi fails in installing the car directly. Let's try to find alternative scenarios to remedy this situation. That will also let us see how urpmi behaves with different pre-existing installations.

Scenario 1 - Trying to install tyre-1 first

Knowing the best solution in advance, let's start by installing tyre-1 and see if it helps urpmi to find this optimum installation.

```
# urpmi tyre-1
installing tyre-1-0.i586.rpm from /home/EDOS/RPMS.car/.
Preparing...
       1/1: tyre                  warning: user prrt does not exist - using root
# urpmi car
To satisfy dependencies, the following 7 packages are going to be installed (0 MB):
car-2-0.i586
door-2-0.i586
engine-2-0.i586
glass-2-0.i586
tyre-2-0.i586
wheel-3-0.i586
window-2-0.i586
Is this OK? (Y/n)
installing tyre-2-0.i586.rpm wheel-3-0.i586.rpm door-2-0.i586.rpm glass-2-0.i586.rpm
window-2-0.i586.rpm engine-2-0.i586.rpm car-2-0.i586.rpm from /home/EDOS/RPMS.car/.
Installation failed:
        tyre = 2 conflicts with glass-2-0.i586
```

It does not solve the problem at all, because urpmi tries to be too smart by upgrading tyre to the latest version, and this is just what we intended to avoid. Unfortunately, it does try to upgrade tyre here and it fails to install again, for the same reason as before.

Scenario 2 - Trying to install window-1 first

urpmi did not use our clue when we installed tyre-1 first, let's see what it does when we try to install window-1 first, and then the car.

```
# urpmi window-1
To satisfy dependencies, the following 2 packages are going to be
installed (0 MB):
```

```
glass-1-0.i586
window-1-0.i586
Is this OK? (Y/n)
installing glass-1-0.i586.rpm window-1-0.i586.rpm from /home/EDOS/RPMS.car/.
Preparing...
      1/2: glass             warning: user prrt does not exist - using root
      2/2: window            warning: user prrt does not exist - using root
# urpmi car
To satisfy dependencies, the following 5 packages are going to be installed
(0 MB):
car-2-0.i586
door-2-0.i586
engine-2-0.i586
tyre-2-0.i586
wheel-3-0.i586
Is this OK? (Y/n)
installing tyre-2-0.i586.rpm wheel-3-0.i586.rpm door-2-0.i586.rpm
engine-2-0.i586.rpm car-2-0.i586.rpm from /home/EDOS/RPMS.car/.
Preparing...
      1/5: engine            warning: user prrt does not exist - using root
      2/5: door              warning: user prrt does not exist - using root
      3/5: tyre              warning: user prrt does not exist - using root
      4/5: wheel             warning: user prrt does not exist - using root
      5/5: car               warning: user prrt does not exist - using root
```

Success! This time urpmi installs everything and it reaches the optimum installation (figure **??**). With reason it did not try to upgrade the window.

Scenario 3 - Trying to install wheel-2 first

Let's repeat the experiment, this time by installing wheel-2 first.

```
# urpmi wheel-2
installing wheel-2-0.i586.rpm from /home/EDOS/RPMS.car/.
Preparing...
      1/1: wheel                warning: user prrt does not exist - using root
# urpmi car
To satisfy dependencies, the following 5 packages are going to be installed (0 M
car-2-0.i586
door-2-0.i586
engine-2-0.i586
glass-2-0.i586
window-2-0.i586
Is this OK? (Y/n)
```

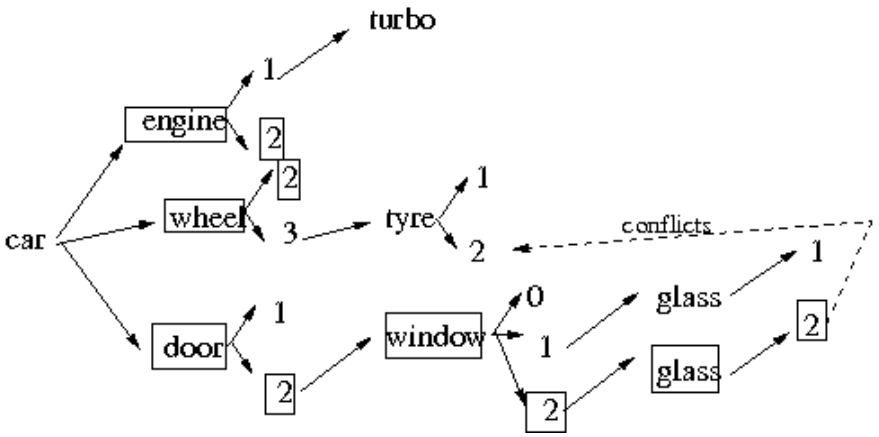


Figure 7.9: Car/Glass: Graph of the installation obtained by urpmi with Scenario 3.

```
installing door-2-0.i586.rpm glass-2-0.i586.rpm window-2-0.i586.rpm engine-2-0.i586.rpm
car-2-0.i586.rpm from /home/EDOS/RPMS.car/.
Preparing...
      1/5: engine               warning: user prrt does not exist - using root
      2/5: glass                warning: user prrt does not exist - using root
      3/5: window               warning: user prrt does not exist - using root
      4/5: door                 warning: user prrt does not exist - using root
      5/5: car                  warning: user prrt does not exist - using root
```

Another success, but this time urpmi did not reach the optimum installation as shown in figure **??**. It detected a conflict with the wheel-3 and tyre-2 and it did not backtrack to find a better solution with tyre-1 instead.

7.6.6 Conclusions on Urpmi

urpmi finds the same solutions as smart in the first attempt when it is given the complete list of RPM packages as command line parameters. However, it fails when relying on the repository hdlist only. When starting with a partially installed environment in Scenarii 2 and 3, it has a good solution comparable to smart.

For the overall problem, urpmi seems to have the same performance of apt. It can not backtrack and solve the conflict problem.

Interestingly, in the alternative scenarios explored with different pre-existing installations, urpmi out-performs smart, however it does only find the optimum solution once out of 3, and it fails in one situation.

As it is evident in this presentation, Urpmi does not try to be complete, as a depsolver, and for this reason it cannot be used for maintaining distributions on the server side, where we need to check installability of every single package.

7.7 Conclusions

The problem of finding an optimal installation candidate, w.r.t. some criteria, is computationally hard and is treated differently by different tools.

Some rely on special heuristics, like Apt, Portage and Urpmi, that perform reasonably well on well-behaved repositories, and ensure that an answer will be reported in a limited time, but at the price of giving up completeness, which means failing to find an installation candidate when it is located too far from the solution suggested by the heuristics.

Others, like Smart, strive to be complete, and really try to explore the solution space, using some special heuristic to try and limit the effect of the combinatorial explosion of the solution space, but at the price of having unacceptably high computation times on some cases; as we have seen, the solution found is not necessarily always optimal either.

We conclude that none of these tool is adapted to checking installability on the server-side, to guarantee quality of a distribution, either because of incompleteness (Apt, Portage, Urpmi), or because of complexity (Smart), and we have hence built our own tools, which are soundly based on formal methods, are complete and perform extremely well on the real-world cases.

The content of this chapter should in no way be construed as a criticism of the tools we analysed: they do try to solve a much harder problem than ours, and they really try their best at it, handling all the added complexity of the extra bits of metadata associated to package management: we know of no satisfactory solution for this problem up to now, and our tools are *not* a replacement for Apt, Protage, Urpmi, Smart and the like.

Nevertheless, we do hope that the detailed analysis presented here will help the designers of future generation client-side package management meta-tools in their quest of the best tradeoff between efficiency and completeness.

Chapter 8

Tools and software currently delivered by the WP2 project team.

As we have clearly demonstrated in the previous chapter, no mainstream existing tool we are aware of can be satisfactorily employed to efficiently perform the static analysis of large package repositories that we set out to do.

In this chapter, we present the tools and the algorithms we have developed, to perform the analysis and the result of some benchmarks, which are extremely satisfactory.

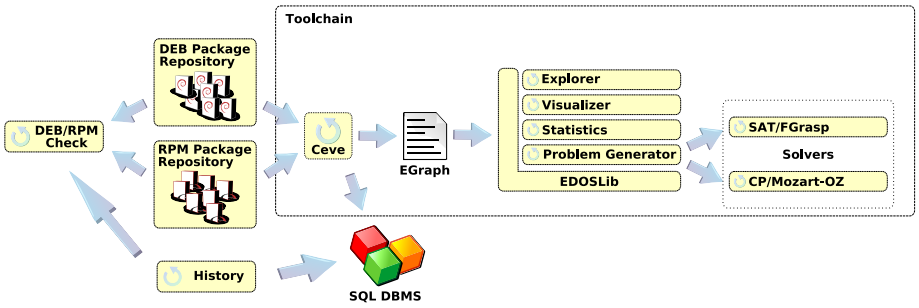


Figure 8.1: The framework

8.1 The framework

We have built a framework that can be used by a *distribution editor* to asses the quality of its distribution and to track problems concerning the underlying package repository (i.e., broken packages in non trimmed repositories). Actually we have two distinct sets of tools (Figure 8.1): a set of independent elements (i.e.,the

toolchain) that executes the analysis by incrementally processing the data collected from a repository; and a very specialized tool that does the analysis in one step.

The rationale for having these two sets of tools is that we want to be able to use the data for other kind of analyses as well. Having a modularized toolchain enables us to reuse part of it even for other purposes. On the other hand, having a specialized, small and efficient tool that performs this particular kind of analysis is good for those *distribution editors* that do not need other kind of features.

Moreover, having plenty of independent tools that execute the same kind of analysis allowed us to validate the whole approach by comparing the results obtained from these different sources.

8.2 The toolchain

In this section we detail the utilities that comprise the toolchain part of the framework. Table 8.2 lists these, along with their kind (CLI for tools with command-line interfaces, LIB for software libraries, and GUI for tools with graphical user interfaces), programming language, code size (in number of words as reported by `wc -w`) and license.

Tool	Kind	Language	Source size (words)	License
ceve	CLI	Ocaml	15k	GPL
EDOSLib	LIB	Java	12k	LGPL
ProblemGenerator	CLI	Java	5k	GPL
EDOS Explorer	CLI	Java	3k	GPL
EDOS Visualizer	GUI	Java	4k	GPL
EDOS Statistics	CLI	Java	1k	GPL
CP/Mozart-Oz Solver	CLI	Mozart-Oz	2k	GPL
SAT transcoder	CLI	Ocaml	6k	GPL
Naive	CLI	Ocaml	5k	GPL
debcheck	CLI	Ocaml	20k	GPL
history	CLI	Ocaml	35k	GPL

Table 8.1: Utilities of the EDOS toolchain.

8.2.1 Ceve

Ceve, the first element of the toolchain, is a generalized package parser. It can read several package formats (most importantly RPM and DEB packages and distributions), but also the XML rpm-metadata format [29]), and output their metadata in several different formats, of which the most important is the EGraph format described in the next section. Ceve is written in OCaml, and it uses the CDuce [28] language for XML input and output.

The full range of output formats that Ceve can produce is:

- The EGraph format;
- The rpm-metadata format;
- A pretty-printing format (a list of all metadata, mostly useful for debugging);
- A dependency graph in the graphviz format;
- A conjunctive normal form format that can be read by the FGrasp solver;
- The SQL format used by rpmfind.

The function of Ceve in the toolchain is that of parsing package metadata from all sorts of formats into one common format (the EGraph format). Since there are many differences between the various package formats, Ceve cannot simply read metadata and output them; the data has to be manipulated. The most notable example of this is RPM dependencies; RPM cannot declare dependencies on other packages directly, but only by way of features. Ceve can resolve these indirect dependencies, so that if package A declares a dependency on feature F, and feature F is provided by packages B and C, in the output package A will need either package B or package C. Also, packages that install different files in the same location conflict with each other, but this is not explicitly declared; Ceve can explicitly add these conflicts to the output.

8.2.2 EDOSLib

EDOSLib is a Java library that provides the foundation for some of the tools that are used in the framework. In particular *EDOSLib* implements: an object model for representing package repositories information and their structure; a set of functionalities to explore manage the package repository struture (e.g., extracting subrepositories and dependency closures); *EGraph* input/output functionalities.

The *EDOSLib* is a concrete implementation of the abstraction that have been presented in the formalization given in Section 6.1.

A reference and the documentation of the *EDOSLib* can be found in the Appendices of this document.

8.2.3 The EGraph package repository description format

The *EGraph* file format is a package-format-agnostic representation of the information that is encoded in the packages stored in a repository. It is based on the XML [10] language and in particular it complies with the *GraphML* [9] specification. This intermediate format has been conceived in order to have a common and uniform representation of the metadata that can be used as input by all the tools of our framework, without having to develop a filter for each package format.

The *GraphML* format is well suited for this purpose because the dependency relationships among packages are easily represented by using graphs and, moreover,

it can be extended in order to accommodate other significant metadata. Figure 8.2 shows an excerpt of the EGraph format. `node` tags specify units (Definition 1) and the associated `data` explicit the available versions, thus the actual packages and, for each version, the `provided` virtual packages.

Dependencies are specified using `edge` tags. `source` and `target` attributes defines the units between the dependency is established, while the `type` attribute defines the dependency type (i.e., `run` or `conflict`). A `version` tag specialize the information by giving, in case, version constraints on the dependency requirement.

```
<node package="aewm" id="aewm">...
  <version number="1.2.0-1"/>
  <provides version="1.2.0-1"
            target="x-window-manager"/>...

<edge source="aewm" target="libc6"
      type="run">...
<version  source="1.2.0-1" operator="ge"
            target="2.2.4-4"/>...
```

Figure 8.2: Excerpt of an EGraph file

The *EGraph* format is an effort for proposing a new metadata format that can complement and extend the existing metadata specified at the package level in order to perform more complicated and effective checking on package repositories, as described in the previous deliverable [15].

8.2.4 ProblemGenerator

ProblemGenerator is an *EDOSLib*-based preprocessor that takes as input the *EGraph* format representing a package repository and gives an encoding of the installability problem in order to verify if the repository is trimmed or not. In particular *ProblemGenerator* performs three steps:

- Extract the package subrepository underlying the dependency closure for each package that has to be analyzed.
- Map the standard package version numbers to integers, as mentioned in Section 6.1; This mapping is necessary in order to reason on dependency relationships with integer-specified-constraints.

Moreover, while generating CP problems we must take care of a problem that may occur when dealing with Virtual Packages or Feautres (i.e., package information specified through a <`provides`> tag). In some package repositories (Debian's above all) it is allowed for a virtual package to have the same name of a

standard package. Usually this happens to "semantically-related" packages. For example `mailx`, which is a mail user agent package is also provided as a virtual package by `mailutils` which happens to be the GNU version of the *mailx* user agent[1]. This poses some problems when setting up CP problems because if not properly handled can lead to inaccurate and confusing constraint (due by the inherent ambiguity of a given identifier, is it a standard or a virtual package?) In order to do so, before generating the CP problem the Problem Generator:

- Expands all the virtual package references which appear in some dependency list of some package to a list of alternatives, made of all the packages (and their associated versions) which provides that virtual package.
- Processes in a different way the generation of those expansions:
  - If the virtual package reference appears in a *depend* relationship we generate an "or" dependency list.
  - If the virtual package reference appears in a *conflict* relationship, we generate an "and" dependency list. This is done because an alternative conflict doesn't make sense.
- Takes care of removing, from a dependency list, any reference to the package associated with the name of the one the list belongs to. This is necessary because of possible name clashes between the standard and virtual packages. For example a package `P` providing `VP` might have a reference `VP` inside one of its dependency lists[2]. After the virtual package references expansion, this will generate self dependencies which can be problematic. For example in a conflict list where the package `P` providing `VP` conflicts with `VP` the package would conflict with itself.

The output is in the format suitable to be processed by the solvers that will perform the actual verification.

8.2.5 EDOS Explorer

The *EDOS Explorer* is an *EDOSLib*-based command line utility that allows the user to effectively navigate in a package repository and to perform operations on it. Actually *EDOS Explorer* is a command line shell that exposes the functionalities of the *EDOSLib*.

EDOS Explorer supports the handling of several package repositories at the same time, with the possibility of switching among them in order to perform different operations. The supported command are summarized in the following table:

[1] There are more than 400 clashing packages in the whole Debian Pool.

[2] Actually this could and should happen only in conflict lists. In fact, a package `P` providing `VP` and depending on `VP` doesn't make too much sense

Command	Description
`load`	Load a package repository
`save`	Save the current package repository
`list`	List the available package repositories
`listpackages`	List all the packages in the current package repository
`info`	Show information about a given package
`whatprovides`	Show the provided features of a given package
`whoprovides`	Show the packages providing the given feature
`getrelations`	Show the dependency relationships of a given package
`showunmet`	Show the unmet dependencies in the package repositories
`extract`	Extract the package cone with respect a given package
`diff`	Show the differences between the current and the given package repository

8.2.6 EDOS Visualizer

The *EDOS Visualizer* is another *EDOSLib*-based utility that allows the user to explore in a graphical way a package repository and visually represent its complexity through the representation of the inter-dependencies among packages.

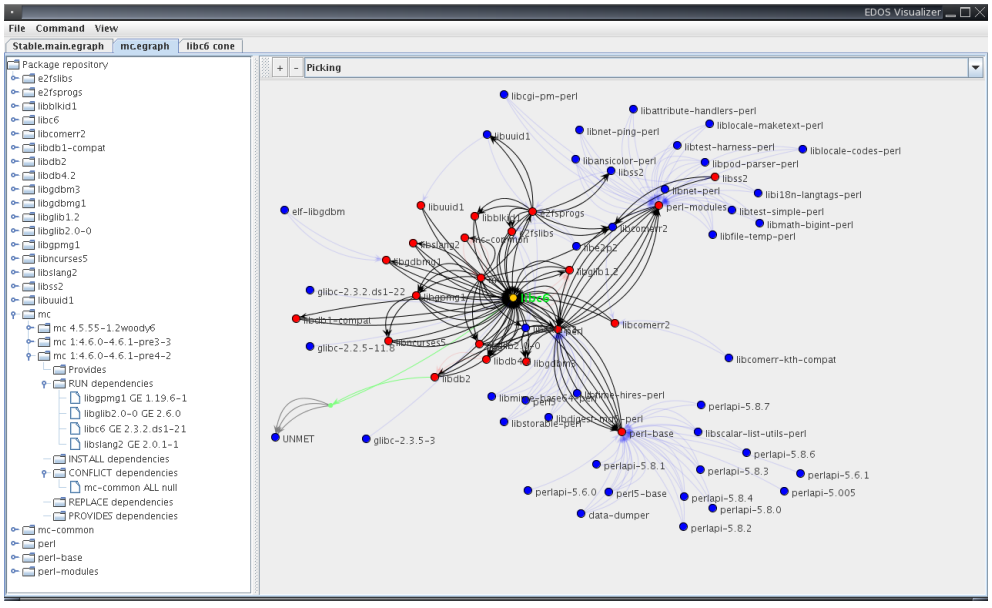


Figure 8.3: The EDOS Visualizer utility

Figure 8.3 shows The *EDOS Visualizer* window. Multiple package repositories can be loaded and managed in different tabs. The left part of each tab shows in a tree-oriented representation all the units present in the repository and, for every unit all the packages belonging to that unit.

By expanding the tree nodes it is possible to obtain all the information regarding the selected package: its provided features and its dependency requirements.

The right part of the tab, on the other hand, shows the graphical representation of the dependency graph. Units are represented by graph vertices and relationships among packages in the different units are represented by graph edges.

The Different colors of the graph elements represent the different kinds of elements in the package repository:

Graph element	**Color**	**Package repository element**
Vertex	Red	Unit
Vertex	Blue	Virtual package
Vertex	Green	A dummy vertex used to model alternative dependency relationships
Edge	Black	Standard dependency (RUN or INSTALL)
Edge	Blue	Provided by relationship
Edge	Red	Conflict dependency
Edge	Gray	Unmet dependency

The *EDOS Visualizer* allows to access to all the functionalities of the *EDOSLib*, in particular package repositories loading, saving and cone extraction. The extracted cone are presented as new package repositories in new tabs.

Moreover the *EDOS Visualizer* provides some "highlighting" functionalities that allows the user to better visualize the structure of the package repository dependency graph:

- Selected vertices highlighting
- Neighborhood highlighting (i.e., the immediate connected units with a dependency relationship). By iterating this kind of highlighting several times it is possible to incrementally show the package cone at a given depth.
- Standard dependency highlighting.
- Conflict highlighting.
- Package cone highlighting.

The *EDOS Visualizer*, finally, allows the user to interact with the dependency graph visualization by performing several operation on it, such as zooming, rotation, vertex selection and displacement.

8.2.7 EDOS Statistics

EDOS Statistics uses the *EDOSLib* in order to create some useful statistics about the structure of a given package repository. In particular, given a package repos-

itory and its dependency graph, *EDOS Statistics* computes the following parameters:

- Total number of vertices (i.e., units and virtual packages)
- Total number of units
- Total number of virtual packages
- Total number of edges (i.e., dependency and provided by relationships)
- For each relationship type, the total number of dependency edges of that type (i.e., the total number of the relationships of a given type among packages)
- Total number of unmet dependencies
- For each relationship type and for each unit, the in-degree for a given edge type (i.e., the number of relationships of a given type having as target a package with in a given unit)
- For each relationship type and for each unit, the out-degree for a given edge type (i.e., the number of relationships of a given type having as source a package with in a given unit)

The statistics are stored in a text file and can be post-processed by using text oriented tools.

8.2.8 CP/Mozart solver

The *CP/Mozart-Oz Solver* translates the output of *ProblemGenerator* to a CP problem as described in section 6.3.2, then solves it using a solver written in the Mozart-Oz language [4] . The solver uses a custom branch-and-bound strategy programmed in Oz itself. While effective on small to medium-sized installability problems, the solver tends to exhibit exponential divergence on large problems.

8.2.9 SAT transcoder

In a first prototyping phase, we converted Oz encodings obtained from the toolchain into DIMACS SAT format (a standard and simple CNF format for SAT solvers) and solved these using external solvers. The *SAT* transcoder takes as input the encoding of an installability problem as produced by *ProblemGenerator* and translates it to a boolean formula as outlined in section 6.3.1 and outputs that formula to a file in conjunctive normal form in the industry-standard DIMACS format. This file can then be fed to a standard SAT solver. These solvers return a minimal set of assignments that satisfy the formula, which the transcoder translates back into minimal lists of packages that are required to enable the installation of the initial package. We only tried two solvers, as there are not many free SAT solvers available. We

first used the GRASP solver before writing the `naive` solver and then switching to the integrated `debcheck` solver. GRASP (an acronym of Generic seaRch Algorithm for the Satisfiability Problem) [21] from T. U. Lisbon seemed to be an adequate first candidate.

8.2.10 Naive solver

After validating the basic encoding concept with Grasp, we implemented a simple solver in Ocaml. The `naive` solver is a straightforward, unoptimized implementation (by the WP2 team) of the well-known Davis-Putnam SAT resolution algorithm. It was written to assess the degree of sophistication needed for a SAT-solver to handle the dependency problems.

8.2.11 The integrated checker: `debcheck/rpmcheck`

One important problem was remaining. When a package p is found to be installable, a set of co-installable packages containing p is furnished by the SAT-solvers. The co-installability can be easily verified (as this is the essence of NP-completeness). However, when a package is found *not* to be installable, these two solvers do not give a satisfying, human-readable explanation.

The most efficient checker in the suite integrates parsing, encoding, solving and displaying in one tool, that comes in two flavors, `debcheck` for Debian-style repositories, and `rpmcheck` for RPM-based repositories.

This tool parses a `Packages-pool`, or a `hdlist` and checks their conformance to the specifications of the package formats. It then checks the installability of every package. It can give, for each package, either a set of sufficient and non-conflicting packages with which it can be installed, or, if the package is not installable, it can concisely explain each failure by presenting a small unsatisfiable subset of the problem. For instance, when a package is found to be uninstallable, the conflict-driven solver can exhibit a small number of "reasons" explaining why the package is not installable, like the following:

```
The package kino-dvtitler 0.2.0-1.1 is not installable
for the following reasons:

    * kino-dvtitler 0.2.0-1.1 depends on kino 0.80-2
    * kino 0.80-2 conflicts with kino-dvtitler 0.2.0-1.1
```

This solver is also extremely efficient (see section 8.3.1).

The tool reads a package control file its standard input. The names (for instance, `emacsen`) of the packages to be tested should be given on the command line. A specific version of a package can be selected by following the package name with an equals sign and the version of the package to test (for instance, `xemacs21=21.4.17-1`). When no package name is provided, all packages in the control file are tested. Command-line options are:

-check	Double-check the results, that is, verify the abundance and peace conditions on the sets computed by the solver.
-explain	Explain the results
-rules	Print generated rules, that is, display the boolean formula solved.
-failures	Only show packages that fail to install.
-successes	Only show packages that can be installed.
-help	Display the list of options.

8.2.12 The `history` tool: package timeline exploration

Overview

The set of Debian packages is divided by architecture (for instance, `i386` or `powerpc`), by "distribution" (stable, unstable, testing or experimental), and also by component (main, contrib or non-free). This structure is a necessary consequence of technical, maintenance process and legal requirements. That different architectures may require different binaries is obvious. Contrary to claims by some large software vendors, proprietary and open-source software can coexist and packaging non-OSS is a benefit both to OSS users and proprietary software vendors : there are still many areas where the use of non-free software is mandated, for instance by hardware (think of 3D acceleration). The contrib component contains OSS software that supports or depends on non-OSS software in the non-free component. Finally, in the Debian process, packages created or updated by maintainers first enter the testing distribution (after a preliminary check of a few days). If they compile well and create no obvious dependency problems, they migrate after a fixed period of ten days into the unstable distribution. As is well known, every year or so, a new stable distribution is made by selecting packages from unstable. Hence we have multiple sets of package that evolve over time. If we fix the architecture, we may define a *package* as a couple (u, v) where u is a unit name and v is a version number. For instance, $(\texttt{aspell} - \texttt{tl}, \texttt{0.02} - \texttt{5})$ is a package. An *archive* is then a function that maps time to sets of packages[3]

The query language

The `history` tool allows command-line exploration of the Debian metadata using an algebraic query language. Daily metadata is extracted from the archives on `snapshot.debian.net` and stored in a MySQL database. The tool then works as a classic read-evaluate-print loop, that is, expressions or directives are entered by the user and their results printed or saved.

[3]It should be noted that, for consistency reasons, once created, the metadata of a package cannot be changed. If there is any problem with the metadata, a new package must be created, that is, a new version number must be used.

Implementation issues Despite the data being stored in a structured manner and being appropriately indexed, performance of even simple SQL queries (such as finding the set of immediate potential dependencies of a set of packages) is poor. This is partly due to the complex nature of metadata, which is, due to disjunctive dependencies, not a graph but a a hypergraph, whose relational representation requires multiple levels of indirection. Also, it is well-known that standard SQL cannot handle transitive closures. Hence we have opted for using the SQL database only as an off-line storage engine; `history` loads the whole database into RAM, and can then answer complex queries efficiently. Because of advances in the performances of filesystems such as ReiserFS, we are planning to drop the SQL backend and use a simple file-based approach for storage. We now give a very short introduction to the query language. Table 8.2 gives an overview of the available operators.

Datatypes The basic Data types handled by `history` are units, packages, sources, sets of the above, dates, integers and booleans. Care has been taken to provide concise notation for describing these. The basic features of a functional language with strict evaluation are provided. Hence unit names can be written without any quoting, and packages are written as `unit'version`. Thus `aspell-tl'0.02-5` is a package where `aspell-tl` is the unit name. Source packages are similarly written but with a backquote. More complex data types include lists, arrays and functions – functions are written as `x -> expr`.

Variables, assignment and binding Variable names start with a '$' and can be assigned with an expression like `$name <- expression`. Local binding is possible with constructions of the form `let` $\$\texttt{name} = \texttt{expression}_1 \texttt{in} \texttt{expression}_2$.

Representation of historical data We remind the reader that to avoid inconsistencies in metadata, all packaging systems require that the metadata for a given package, that is for a given (unit, version) pair, be a constant. Hence we can represent the evolution of the repositories as functions that map maps dates to sets of packages. As such, archive contents are accessed in `history` by applying the archive ID (an integer that can be obtained with the `#list_archives` directive) and a date to the pre-defined `$archives` function. For instance, `$archives 3 2005-11-02` returns the set of packages contained in `debian/stable/main/binary-i386` on November the 2nd, 2005.

Operations on sets The usual Boolean operations on sets (intersection $\&$, union $|$ and difference $\backslash$) are allowed. Sets can be filtered by regular expressions or user-defined functions. With operators such as `exists` and `for_all`, this effectively provides first-order quantified queries. Besides directives that print the metadata in human-readable form, various operators are provided so that the metadata can be used in complex expressions. For instance, if p is a package, `provides`(p) is the set

of units provided by p, and if s is a set of packages, `closure`(s) is the dependency closure of s : all packages that packages of s might need to run are contained in s – but due to disjunctive dependencies, s will usually contain extraneous packages, and due to conflicts, it might not be possible to install all packages of s.

Integration of the dependency solver The dependency solver of `debcheck` and `rpmcheck` has been integrated into `history` as an operator `install`(p_1, p_2) which computes a set p of co-installable packages such that $p_1 \subseteq p \subseteq p_2$.

Example session We first invoke history. It is assumed that the database server is properly configured.

```
% history -interactive
Total 22172 units in 9 archives.
>
```

We may then list the archives known to the system.

```
> #list_archives
1  debian stable main binary-i386
2  debian stable contrib binary-i386
3  debian stable non-free binary-i386
4  debian testing main binary-i386
5  debian testing contrib binary-i386
6  debian testing non-free binary-i386
7  debian unstable main binary-i386
8  debian unstable contrib binary-i386
9  debian unstable non-free binary-i386
```

We are interested in the unstable/main archive which has number 7. Let's see what packages are available in that archive on February first, 2006.

```
> $archives 7 2006-02-01
{ 3dchess'0.8.1-11.1, 3ddesktop'0.2.9-5.1,
  44bsd-rdist'20001111-6, 6tunnel'0.11rc1-1, 915resolution'0.5-1,
  9base'2-1, 9menu'1.8-1.1, 9wm'1.2-7, a2ps'1:4.13b.dfsg.1-0.1,
  a2ps-perl-ja'1.45-4, aa3d'1.0-4, aap'1.072-1, aap-doc'1.072-1,
  aatv'0.3-2, abakus'0.90-6, abc2ps'1.3.3-3, abcde'2.3.99.3-1,
  abcm2ps'4.11.8-1, abcmidi'20051010-1, ... } : set of package as
int set
```

As there are thousands of packages in that set of packages, we may first assign the set to a variaible, and then count its cardinality.

```
> $x <- $archives 7 2006-02-01
> count($x)
17203 : int as int 17203
```

Let's see what are the packages of $x whose unit name start with ocaml by filtering it with a regular expression.

```
> $x ~ /^ocaml/
{ ocaml'3.09.1-2, ocaml-base'3.09.1-2, ocaml-base-nox'3.09.1-2,
  ocaml-compiler-libs'3.09.1-2, ocaml-core'3.08.0.2,
  ocaml-dbforge'1.9.9.cvs20051129-2+b2, ocaml-findlib'1.1.1-2,
  ocaml-interp'3.09.1-2, ocaml-libs'3.08.0.2, ocaml-mode'3.09.1-2,
  ocaml-native-compilers'3.09.1-2, ocaml-nox'3.09.1-2,
  ocaml-report'1.9.9.cvs20051129-2+b2, ocaml-source'3.09.1-2,
  ocaml-tools'2005.29.04-5, ocaml-ulex'0.8-2,
  ocamlcvs'1.9.9.cvs20051129-2+b2, ocamldsort'0.14.3-1.2,
  ocamlweb'1.37-2 } : set of package as int set
```

Assume we want a very basic Ocaml development system. We will need the ocaml'3.09.1-2 package and a line editor, such as ledit. Let's see which versions of ledit are available:

```
> versions(ledit)
{ ledit'1.11-4, ledit'1.11-6, ledit'1.11-7 } : set of package as int set
```

These are all the versions that are known by the system. To find those that are inside the set $x, we can intersect those two sets:

```
> $x & versions(ledit)
{ ledit'1.11-7 } : set of package as int set
```

The dependency closure of a set of packages x is an upper bound on the set y of packages needed to install x. It is obtained by recursively following the Depends: and Pre-depends: fields.

```
> $y <- closure({ ocaml'3.09.1-2, ledit'1.11-7 })
> count($y)
313 : int as int 313
```

The set $y should satisfy the abundance condition (see section 6.1)

```
> #abundance $y
Abundance condition satisfied.
```

However the set $y is not co-installable because of conflicts:

```
> count(conflicts($y) & $y)
299 : int as int 299
```

Hence we need to invoke the solver to find an installation of ocaml and ledit:

```
> $i <- install({ ocaml'3.09.1-2, ledit'1.11-7 }, $y)
> count($i)
36 : int as int 36
```

A 36-package installation is adequate. We can output the solution to a file in a suitable one-entry-per-line format:

```
> #dump "/tmp/installation" $i
```

The content of that file is:

```
% cat /tmp/installation
binutils'2.16.1cvs20060117-1
coreutils'5.94-1
cpp'4:4.0.2-2
cpp-4.0'4.0.2-10
debconf'1.4.71
debconf-english'1.4.71
debianutils'2.15.3
gcc'4:4.0.2-2
gcc-4.0'4.0.2-10
gcc-4.0-base'4.0.2-10
ledit'1.11-7
libacl1'2.2.35-1
libattr1'2.4.31-1
libc6'2.3.6-3
libc6-dev'2.3.6-3
libgcc1'1:4.0.2-10
libgdbm3'1.8.3-2
libncurses5'5.5-1
libncurses5-dev'5.5-1
libselinux1'1.28-4
libsepol1'1.10-2
libx11-6'6.9.0.dfsg.1-4
linux-kernel-headers'2.6.13+0rc3-2
lsb-base'3.0-15
ncurses-bin'5.5-1
ocaml'3.09.1-2
ocaml-base'3.09.1-2
ocaml-base-nox'3.09.1-3
ocaml-interp'3.09.1-3
ocaml-nox'3.09.1-3
perl-base'5.8.8-2
sed'4.1.4-5
tcl8.4'8.4.12-1
tk8.4'8.4.12-1
x11-common'6.9.0.dfsg.1-4
xlibs-data'6.9.0.dfsg.1-4
```

Operator	Meaning
$s\&t, s\|t, s \setminus t$	Boolean set intersection, union and difference
`provides`(p)	Set of units provided by a package
`conflicts`(p)	Set of packages that conflict with a package
`closure`(p)	Dependency closure of a package
`source`(p)	Source of a package
`unit`(p)	Unit of a package
`latest`(u)	Latest version of a unit
`versions`(u)	All the versions of a unit
`what_provides`(u)	The set of packages that provide a unit
`replaces`(p)	The set of packages replaced by a package
`install`(p, q)	Returns an installation of the set of packages p inside the set q
`member`(x, s)	True when the element `x` is a member of the set `s`
`filter`(s, f)	Return the elements of s for which $f(s)$ is true.
`exists`(s, f)	True when $f(s)$ is true for one element of s
`for_all`(s, f)	True when $f(s)$ is true for all elements of s
`treat`(s, f)	Create an array of values obtained by applying f to elements of s

Table 8.2: Operators. Most operators are overloaded to work on sets. Functions are first-class values with lexical scoping that can be defined anonymously.

8.3 Solvers, complexity analysis and benchmarks

It is really not evident that even the first static analysis of the individual installability of a package in a repository is actually tractable in practice: due to the rich language allowed to describe package dependencies in the mainstream FOSS distributions, even the simplest problems (checking installability of a single package) may involve verifications over a large number of other packages. During our first investigations of these problems, reported in Deliverable D2.1, we have indeed already proven the following complexity result.

Theorem 2 (Package installability is an NP-complete problem) *Checking whether a single package P can be installed, given a repository R, is NP-complete.*

Nevertheless, this strong limiting result does not mean that we will not be able to decide installability and the other problems in practice: the actual instances of these problems, as found in real repositories, can be quite simple in the average when encoded as instances of the boolean satisfiability problem in the conjunctive normal form (see section 6.3.1 for details on the encoding). Our tools perform extremely well on the real-world sized problems on which we have run our tests.

8.3.1 Experimental results

In this section we show some experimental results that we have gathered by analyzing with our tools two of the most famous *GNU/Linux*-based distributions: Debian

GNU/Linux [2] and Mandriva Linux [3]. These experimental results include statistics pertaining to the hardness of the SAT instances related to installability problems, performance measurements of different SAT-solvers and statistics on broken packages.

Size and hardness of SAT instances

Figure 8.4 gives a histogram showing the number of packages as a function of the size of the dependency closure, from the Debian stable, unstable and testing pools on 2005-12-13, which has 31149 packages. The average closure size is 158; 50% of the packages have a closure size of 71 or less, 90% of 372 or less, and 99% of 1077 or less. These numbers show that naive combinatorial algorithms, exponential in the size of the dependency closure, are clearly out of the question.

However there exists an easy to compute indicator of the hardness of a boolean satisfiability problem that is more precise than mere formula size. Any boolean formula can be efficiently converted to a formula in conjunctive normal form with no more than three literals per clause. Such formulae are said to be in 3SAT form. The "temperature" T of a 3SAT formula is defined as $T = m/n$ where m is the number of clauses and n the number of variables and there is strong, widely accepted theoretical and practical evidence that hard SAT problems (once converted to 3SAT) tend to have a temperature close to the "transition temperature" whish is 4.2, while SAT problems with temperatures well below or above that limit are easier to solve. Figure 8.5 gives an historgram of the temperatures of the Boolean formulae generated by our encoding of the installability problems. Their temperatures range from 0.75 to 1.49, well below the transition temperature. This result confirms that we are dealing with relatively easy satisfiability problems, maybe owing to the small-world nature of the dependency graphs This provides additional strong evidence of the fact that the instances of the NP-complete SAT problem that are important to us are easy.

Performance issues and development of SAT solvers

The performance of GRASP was very good, no package needed more than one second of computation time (on a 3.0GHz Xeon). In figure 8.6(a), the execution time of GRASP, measured in seconds, is plotted against the size of the input Oz file encoding the installability of a package. Please note that the Y axis is logarithmic. The data suggests a quadratic relationship between execution time and input size, and a quadratic curve indeed gives a good fit.

The `naive` solver was only about ten times slower than GRASP. This shows that dependency problems do not need very sophisticated SAT-solving techniques. It also exhibited a quadratic behaviour similar to GRASP: the execution times did not exceed ten seconds per package. At this point we were convinced of the practicality of using standard SAT-solving techniques for doing static installability analysis.

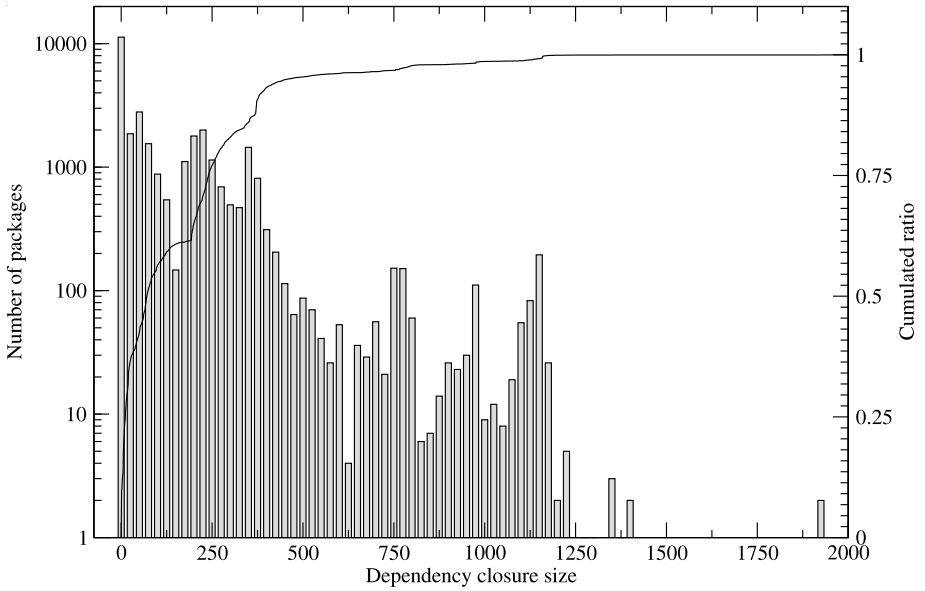


Figure 8.4: Number of packages as a function of the size of their dependency closures.

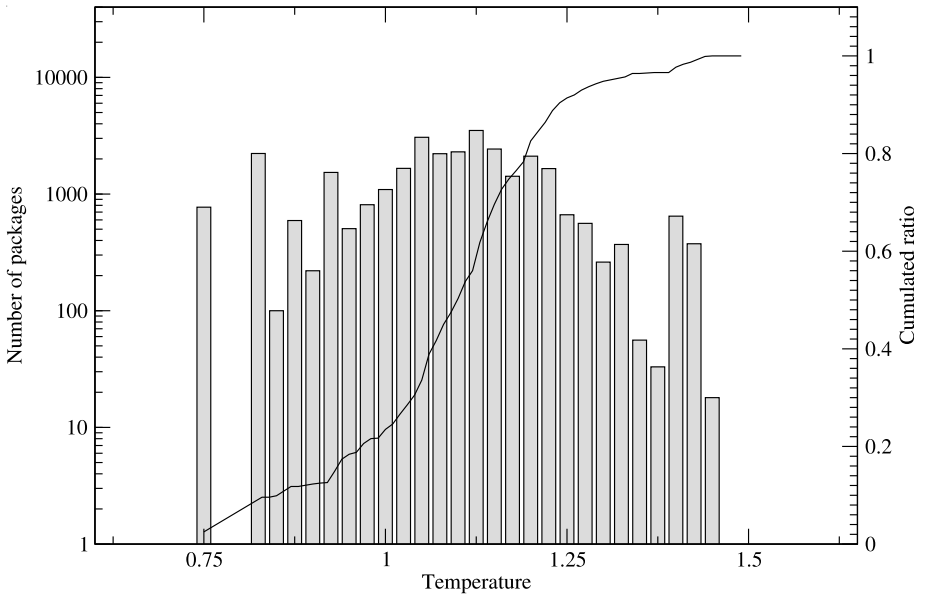


Figure 8.5: Number of packages as a function of the “temperature” of the SAT problems corresponding to their installability problems.

We have searched the literature on SAT-solving for an algorithm that provides some explanation capabilities while being reasonably simple but efficient. In other words, we were looking for a state-of-the-art "simple" SAT solver giving explanations. The `debcheck`/`rpmcheck` solver is based on two papers describing such algorithms [16, 19]. An important aspect of this tool is that works by loading the whole metadata for a pool, computing a giant boolean formula encoding the installability of all the packages, and then testing the satisfiability of the individual packages. While solving the installability of a package, the solver can discover the installability of dependent packages; time is saved by not checking their installability again. This gives an extremely fast solver that can check whole repositories in minutes (compared to the hours that would be required by the other approach), as can be seen in table 8.3.1. This is an indication of how long it takes to complete some operations on our project server, which is a single-processor Intel Xeon 3.4 GHz machine running Mandriva Linux. The figures for rpmcheck/debcheck include time for parsing, the figures for SAT do not.

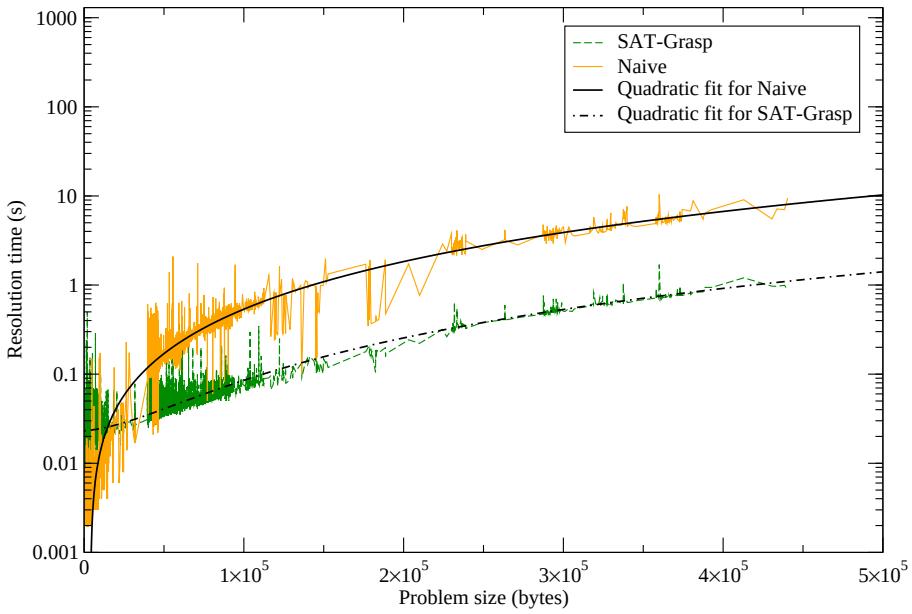
Operation	**User time**
Parsing	
Mandriva 2006	13s
Debian snapshot	30s
Cone extraction	
Mandriva 2006	4m06s
Debian snapshot	27m40s
Installability checks	
Mandriva 2006 with GRASP	55s
Mandriva 2006 with rpmcheck	8s
Debian snapshot with GRASP	18m13s
Mandriva 2006 with debcheck	43s

Table 8.3: Performance of `debcheck` and `rpmcheck`.

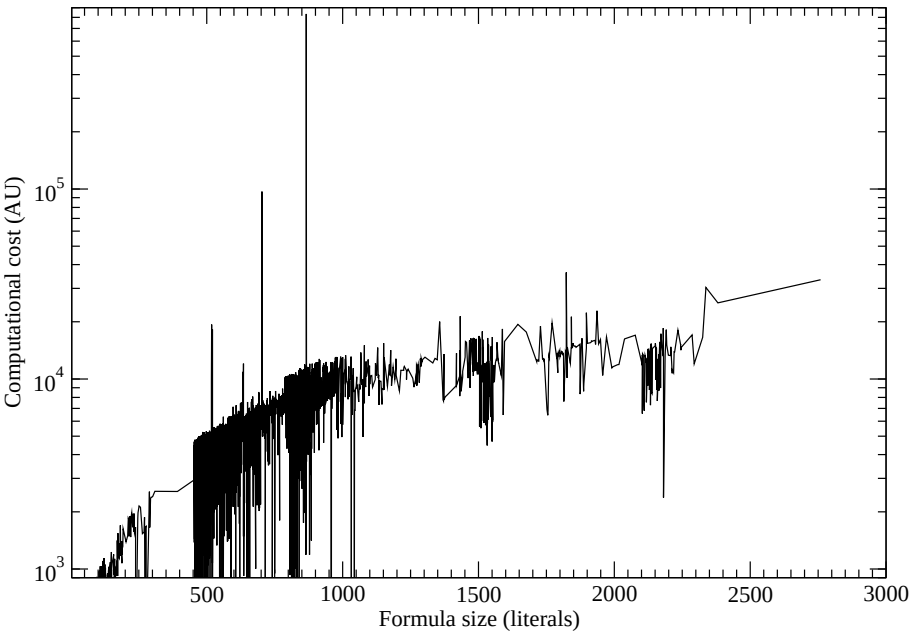
Figure 8.6(b) gives an estimate of the computational cost of this solver against the size of CNF formulae measured in number of literals. As `debcheck` was working too fast, it was not possible to use system timers to get a sufficiently precise measurement of its execution times. We therefore counted the number of function calls (for an adequate subset of the available functions), which should roughly be proportional to execution times. Although the execution time of the solver is quite irregular, it is roughly linear in the formula size for these instances.

Broken packages

The reference package repositories are a snapshot of the complete Debian pool located at `http://ftp.debian.org/pool` and the packages distributed with the Mandriva 2006 Edition.



(a) Performances of `naive` vs. `Grasp` as seconds vs. bytes.



(b) Performance of `debcheck` arbitrary units vs. litterals.

Figure 8.6: Performance of various solvers on the Debian pool. In figure 8.6(a), execution time, measured in seconds, is plotted against the size, in bytes, of the Oz code fragment generated with the toolchain and encoding the installability of packages.

Out of the 4211 packages in the main part of the Mandriva 2006 distribution, 42 are not installable. In 38 cases, this is due to a dependency that is not available; in the case of four packages, `mozilla-thunderbird-enigmail-{de,es,fr,it}`, the problem is a simultaneous dependency and conflict with the package `mozilla-thunderbird-enigmail`.

The Debian snapshot contains 34701 packages, of which 123 are not installable. Here, 111 cases are due to an unavailable dependency, and 12 packages have an inherent conflict (for example, the package `cacti-cactid` version 0.8.6e-2 depends on `libsnmp5`, version 5.2.1.2 or greater; but the only appropriate version of `libsnmp5` in the pool, 5.2.1.2-2, conflicts with all versions of `cacti-cactid` up to and including 0.8.6e-2).

Chapter 9

Conclusions

We have done an extensive analysis of the whole set of problems that are in the focus of WorkPackage 2, ranging from upstream tracking, to thinning, rebuilding, and dependency managements for F/OSS distributions.

For the dependency management problems, we have also provided a whole set of industry-strength tools that have already been incorporated into the production chain of Caixa Magica, and are in the process of being incorporated in the Mandriva one.

We plan to disseminate widely these tools and make them known to the Debian community too.

Formal methods at work

We want to particularly stress here the fact that the tools that have been built and engineered by the team are really based on a sound formal theoretical foundations. In particular, we have

- formally defined the installability problem and all its related notions, like dependency closure, subrepository etc.
- formally proved its NP-completeness
- formally provided an encoding of the installability problem both into a Finite Domain Constraint Problem, and into a SAT problem
- developed several independent implementations of the verification technique, one fully integrated, in the `debcheck/rpmcheck tool`, and one based on a

modularized toolchain, able to call either an Oz-based CP solver, or various SAT solver (one custom made, and one mainstream, the `fgrasp` solver developed in Portugal).
All of these different tools produce now the very same results, and the availability of each of them has been essential, as by comparing the results of the earlier versions it has been possible to spot and repair various implementation mistakes, and to clarify doubts and ambiguity appearing in the DEB or RPM package formats.

While we did not do a machine-checked proof of correctness of these tools, which is way beyond the scope of the current project, and would require far more resources, we are now extremely confident in the soundness and completeness of the results provided by our tools.

This is a huge step forward w.r.t. preexisting tools, and we believe to have contributed to a significant advance in the state of the art.

Ongoing work

A significant amount of work has been done, but we discovered a wide area for future investigation, and development. It is clear that the research area pioneered by this project has opened up new perspectives, and we will need years of study and dissemination to explore them.

Bibliography

[1] Debian autobuilder. http://buildd.debian.org.

[2] Debian GNU/Linux. http://www.debian.org.

[3] Mandriva Linux. http://www.mandriva.com.

[4] The Mozart programming system. http://www.mozart-oz.org.

[5] The new mandriva build system. http://qa.mandriva.com/twiki/bin/view/Main/RepositorySystem.

[6] E. C. Bailey. Maximum RPM, taking the Red Hat package manager to the limit. http://rikers.org/rpmbook/,http://www.rpm.org, 1997.

[7] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information flow based event distribution middleware. *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, 1999.

[8] G. Beekmans. Linux From Scratch. http://www.linuxfromscratch.org.

[9] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. Graphml specification, 2002. http://graphml.graphdrawing.org/specification.html.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M. F. Yergeau. Extensible markup language (xml) 1.0 (third edition), 2004. http://www.w3.org/TR/REC-xml.

[11] M. Broy and E. Denert. *Software Pioneers: Contributions to Software Engineering*. Springer-Verlag, 2002.

[12] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems 19(3)*, pages 332–383, 2001.

[13] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC, 20(8)*, October 2002.

[14] Debian Group. Debian policy manual. http://www.debian.org/doc/debian-policy/, 1996–1998.

[15] EDOS Project Workpackage 2 Team. Deliverable 1, 2005. http://www.edos-project.org/xwiki/bin/Main/Deliverables.

[16] N. Eén and N. Sörensson. An extensible SAT-solver. In Giunchiglia and Tacchella [19], pages 502–518.

[17] D. Eklund. The lib update/autoupdate suite. http://luau.sourceforge.net/, 2003–2005.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.

[20] Mandriva. URPMI. http://www.urpmi.org, 2005.

[21] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:5:506–521, 1999.

[22] G. Niemeyer. Smart package manager. http://labix.org/smart, 2005.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Middleware'01*, 2001.

[24] G. N. Silva. Apt-howto. http://www.debian.org/doc/manuals/apt-howto, 2004.

[25] S.Zhuang, B.Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. *In NOSSDAV.2001*, June 2001.

[26] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

[27] L. Taylor and L. Tuura. Ignominy: a tool for software dependency and metric analysis with examples from large HEP packages. In *Proceedings of CHEP'01*, 2001.

[28] The CDuce Team. Cduce. http://www.cduce.org.

[29] The RPM-Metadata Project. Xml package metadata. http://linux.duke.edu/projects/metadata.

[30] L. A. Tuura. Ignominy: tool for analysing software dependencies and for reducing complexity in large software systems. In *Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, volume 502, pages 684–686, 2003.

[31] T. van der Storm. Variability and component composition. In *Proceedings of the Eighth International Conference on Software Reuse (ICSR-8)*, 2004.

Appendix A

Package org.edos_project.model.util

Package Contents *Page*

Classes

A.1 Classes

A.1.1 CLASS **PackageDependencyData**

A data bundle containing all the information for building a dependency relationship

DECLARATION

public class PackageDependencyData
extends java.lang.Object

CONSTRUCTORS

- *PackageDependencyData*
  `public` **PackageDependencyData**`(`
  `org.edos_project.model.DependencyType` **dependencyType** `)`
- *PackageDependencyData*
  `public` **PackageDependencyData**`(`
  `org.edos_project.model.DependencyType` **dependencyType**`,`
  `org.edos_project.model.VersionRelationship`
  versionRelationship `)`
- *PackageDependencyData*
  `public` **PackageDependencyData**`( java.lang.String`
  dependencyTypeString `)`

METHODS

- *getDependencyType*
  `public DependencyType` **getDependencyType**`( )`
- *getOperator*
  `public VersionOperator` **getOperator**`( )`
- *getSourcePackageUnit*
  `public String` **getSourcePackageUnit**`( )`
- *getSourcePackageVersion*
  `public String` **getSourcePackageVersion**`( )`
- *getTargetPackageUnit*
  `public String` **getTargetPackageUnit**`( )`
- *getTargetPackageVersion*
  `public String` **getTargetPackageVersion**`( )`
- *setOperator*
  `public void` **setOperator**`( java.lang.String` **operatorString** `)`

- *setOperator*
  `public void` **setOperator**`(`
  `org.edos_project.model.VersionOperator` **operator** `)`
- *setSourcePackageUnit*
  `public void` **setSourcePackageUnit**`( java.lang.String`
  sourcePackageUnit `)`
- *setSourcePackageVersion*
  `public void` **setSourcePackageVersion**`( java.lang.String`
  sourcePackageVersion `)`
- *setTargetPackageUnit*
  `public void` **setTargetPackageUnit**`( java.lang.String`
  targetPackageUnit `)`
- *setTargetPackageVersion*
  `public void` **setTargetPackageVersion**`( java.lang.String`
  targetPackageVersion `)`

A.1.2 CLASS **PackageDependencyDataSet**

A data bundle containing all the information needed to build an alternative dependency

DECLARATION

public class PackageDependencyDataSet **extends** java.util.HashSet

SERIALIZABLE FIELDS

- private DependencyType dependencyType
  –

CONSTRUCTORS

- *PackageDependencyDataSet*
  `public` **PackageDependencyDataSet**`(`
  `org.edos_project.model.DependencyType` **dependencyType** `)`

METHODS

- *add*
  `public boolean` **add**`(`
  `org.edos_project.model.util.PackageDependencyData`
  packageDependencyData `)`
- *getDependencyType*
  `public DependencyType` **getDependencyType**`( )`

A.1.3 CLASS **Type**

Utility class to check the type of dependency graph elements

DECLARATION

public class Type
extends java.lang.Object

CONSTRUCTORS

- *Type*
  `public` **Type**`( )`

METHODS

- *isAlternativeEdge*
  `public static boolean` **isAlternativeEdge**`( java.lang.Object` **obj** `)`
- *isAlternativeVertex*
  `public static boolean` **isAlternativeVertex**`( java.lang.Object` **obj** `)`
- *isStandardPackageEdge*
  `public static boolean` **isStandardPackageEdge**`( java.lang.Object` **obj** `)`
- *isStandardUnitVertex*
  `public static boolean` **isStandardUnitVertex**`( java.lang.Object` **obj** `)`
- *isVirtualPackageEdge*
  `public static boolean` **isVirtualPackageEdge**`( java.lang.Object` **obj** `)`
- *isVirtualPackageVertex*
  `public static boolean` **isVirtualPackageVertex**`( java.lang.Object` **obj** `)`

A.1.4 CLASS **VertexColorLabeller**

Utility class used for extracting package cones.

DECLARATION

public class VertexColorLabeller
extends java.lang.Object

FIELDS

- public static final String DEFAULT_KEY

  –

CONSTRUCTORS

- *VertexColorLabeller*
  `public` **VertexColorLabeller**`( Graph` **graph** `)`
- *VertexColorLabeller*
  `protected` **VertexColorLabeller**`( Graph` **graph**`,`
  `java.lang.Object` **key** `)`

METHODS

- *clear*
  `public void` **clear**`( )`
- *getColor*
  `public VertexColorLabeller.ColorType` **getColor**`( Vertex` **v** `)`
- *getColorLabeller*
  `public static VertexColorLabeller` **getColorLabeller**`( Graph`
  graph `)`
- *getColorLabeller*
  `public static VertexColorLabeller` **getColorLabeller**`( Graph`
  graph`, java.lang.Object` **key** `)`
- *getGraph*
  `public Graph` **getGraph**`( )`
- *hasColorLabeller*
  `public static boolean` **hasColorLabeller**`( Graph` **graph** `)`
- *hasColorLabeller*
  `public static boolean` **hasColorLabeller**`( Graph` **graph**`,`
  `java.lang.Object` **key** `)`
- *setColor*
  `public void` **setColor**`( Vertex` **v**`,`
  `org.edos_project.model.util.VertexColorLabeller.ColorType`
  color `)`

A.1.5 CLASS **VertexColorLabeller.ColorType**

DECLARATION

public static final class VertexColorLabeller.ColorType
extends java.lang.Enum

FIELDS

- public static final VertexColorLabeller.ColorType GRAY

  –

- public static final VertexColorLabeller.ColorType BLACK

  –

METHODS

- *valueOf*
  `public static VertexColorLabeller.ColorType` **valueOf**`(`
  `java.lang.String` **name** `)`
- *values*
  `public static final VertexColorLabeller.ColorType` **values**`(`
  `)`

Appendix B

Package org.edos_project.model

Package Contents *Page*

Classes

This class models a virtual package vertex

B.1 Classes

B.1.1 CLASS **AlternativeEdge**

An alternative edge models one of the requirements of an alternative dependency of the type D1 — D2 — D3. <p/>An alternative edge connects standard package vertices to alternative vertices that are used to simulate hyperedges (i.e., one-to-many relationships)

DECLARATION

public class AlternativeEdge
extends org.edos_project.model.DependencyEdge

CONSTRUCTORS

- *AlternativeEdge*
  `public` **AlternativeEdge**`(`
  `org.edos_project.model.DependencyType` **dependencyType**`,`
  `org.edos_project.model.StandardUnitVertex` **source**`,`
  `org.edos_project.model.AlternativeVertex` **target** `)`
  - **Usage**
    - Initialize the alternative edge
  - **Parameters**
    - `dependencyType` - The dependency type
    - `source` - The source vertex (must be a StandardUnitVertex)
    - `target` - The target vertex (must be an AlternativeVertex)

B.1.2 CLASS **AlternativeVertex**

An alternative vertex is a fictious vertex that is used to simulate hyperedges (i.e., one-to-many relationships). An alternative vertex has one in-edge coming from a standard package vertex and several out-edges, each one for every alternative. <p/>So if there is a dependency from package A towards packages D1 — D2 — D3 this will be modeled by an edge from the package vertex A to an alternative vertex K and three edges from K to package vertices D1, D2 and D3.

DECLARATION

public class AlternativeVertex
extends org.edos_project.model.DependencyVertex

CONSTRUCTORS

- *AlternativeVertex*
  `public` **AlternativeVertex**`( )`

METHODS

- *getStandardDependencyEdges*
  `public Set` **getStandardDependencyEdges**`( )`
  - **Usage**
    - Get all the alternatives associated to this alternative vertex
  - **Returns** - A set with all the alternative edges starting from this edge
- *toString*
  `public String` **toString**`( )`

B.1.3 CLASS DependencyEdge

A dependency edge models a dependency between packages represented by package vertices.

DECLARATION

public abstract class DependencyEdge
extends DirectedSparseEdge

CONSTRUCTORS

- *DependencyEdge*
  `public` **DependencyEdge**`(`
  `org.edos_project.model.DependencyType` **dependencyType**`,`
  `org.edos_project.model.DependencyVertex` **source**`,`
  `org.edos_project.model.DependencyVertex` **target** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `dependencyType` - The dependency type
    - `source` - The source package vertex
    - `target` - The target package vertex

METHODS

- *getDependencyType*
  `public DependencyType` **getDependencyType**`( )`
  - **Returns** - The dependency type for this edge

B.1.4 CLASS **DependencyGraph**

The dependency graph contains all the dependencies of a package repository

DECLARATION

public class DependencyGraph **extends** SparseGraph

FIELDS

- public static final String UNMET_VERTEX_NAME
  -

CONSTRUCTORS

- *DependencyGraph*
  `public` **DependencyGraph**`(`
  `org.edos_project.model.PackageRepository` **packageRepository**
  `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `packageRepository` - The package repository associated with this dependency graph

METHODS

- *addEdge*
  `public void` **addEdge**`(`
  `org.edos_project.model.util.PackageDependencyData` **pdd** `)`
  - **Usage**
    - Create a dependency edge
  - **Parameters**
    - `pdd` - The data bundle containing dependency information

---

- *addEdge*
  `public void` **addEdge**`(`
  `org.edos_project.model.util.PackageDependencyDataSet`
  pddSet `)`
  - **Usage**
    - Create an alternative edge
  - **Parameters**
    - `pddSet` - The data bundle containing the dependency alternatives information

---

- *addStandardUnitVertex*
  `public StandardUnitVertex` **addStandardUnitVertex**`(`
  `java.lang.String` **name** `)`
  - **Usage**
    - Add a standard unit vertex
  - **Parameters**
    - `name` - The name of the package unit
  - **Returns** - The newly created standard unit vertex

- *addVirtualPackageVertex*
  `public VirtualPackageVertex` **addVirtualPackageVertex**`(`
  `java.lang.String` **name** `)`
  - **Usage**
    - Add a virtual package vertex
  - **Parameters**
    - `name` - The virtual package name
  - **Returns** - The newly created virtual package vertex

- *findStandardUnitVertexByName*
  `public StandardUnitVertex` **findStandardUnitVertexByName**`(`
  `java.lang.String` **name** `)`
  - **Usage**
    - Find a standard unit vertex
  - **Parameters**
    - `name` - The unit name
  - **Returns** - The standard unit vertex or null if not found

---

- *findVertexByName*
  `public UnitVertex` **findVertexByName**`( java.lang.String`
  name `)`
  - **Usage**
    - Find a vertex by name looking up first virtual package vertices and then standard unit vertices.
  - **Parameters**

    - `name` - The name to look for
  - **Returns** - The found vertex or null if nothing has been found

---

- *findVirtualPackageVertexByName*
  `public VirtualPackageVertex` **findVirtualPackageVertexByName**`(`
  `java.lang.String` **name** `)`
  - **Usage**
    - Find a virtual package vertex
  - **Parameters**
    - `name` - The virtual package name
  - **Returns** - The virtual package vertex or null if not found

---

- *getUnmetDependencyVersionRelations*
  `public Set` **getUnmetDependencyVersionRelations**`( )`
  - **Usage**
    - Get all the relationships that are unmet in this dependency graph
  - **Returns** - The unmet relationships set

- *numAlternativeEdges*
  `public int` **numAlternativeEdges**`(`
  `org.edos_project.model.DependencyType` **dt** `)`
  - **Usage**
    - Calculates the number of alternative edges
  - **Returns** - The number of alternative edges

- *numAlternativeVertices*
  `public int` **numAlternativeVertices**`( )`
  - **Usage**
    - Calculates the number of alternative vertices
  - **Returns** - The number of alternative vertices

- *numDependenciesInDegree*
  `public int` **numDependenciesInDegree**`( java.lang.String`
  unitName`, org.edos_project.model.DependencyType` **dt** `)`
  - **Usage**
    - Calculates the number of dependency edges targeting a given unit vertex
  - **Parameters**
    - `unitName` - The unit name
    - `dt` - The dependency type
  - **Returns** - the number of dependency edges targeting the given unit vertex

---

- *numDependenciesOutDegree*
  `public int` **numDependenciesOutDegree**`( java.lang.String`
  unitName`, org.edos_project.model.DependencyType` **dt** `)`

  - **Usage**
    * Calculates the number of dependency edges starting from a given unit vertex
  - **Parameters**
    * `unitName` - The unit name
    * `dt` - The dependency type
  - **Returns** - the number of dependency edges targeting the given unit vertex

---

- *numPackageVertices*
`public int` **numPackageVertices**`( )`

  - **Usage**
    * Calculates the number of unit vertices
  - **Returns** - The number of unit vertices

- *numStandardDependencyEdges*
`public int` **numStandardDependencyEdges**`(`
`org.edos_project.model.DependencyType` **dt** `)`

  - **Usage**
    * Calculates the number of standard dependency edges of a given type
  - **Parameters**
    * `dt` - The dependency type
  - **Returns** - The number of standard dependency edges

- *numUnmetDependencies*
`public int` **numUnmetDependencies**`( )`

  - **Usage**
    * Calculates the number of unmet dependency edges
  - **Returns** - The number of unmet dependency edges

- *numUnmetDependenciesFromAlternatives*
`public int` **numUnmetDependenciesFromAlternatives**`( )`

  - **Usage**
    * Calculates the number of unmet dependency edges starting from an alternative edge
  - **Returns** - The number of unmet dependency edges starting from an alternative edge

---

- *numVirtualPackageEdges*
`public int` **numVirtualPackageEdges**`( )`

  - **Usage**
    * Calculates the number of virtual package edges
  - **Returns** - The number of virtual package edges

- *numVirtualPackageVertices*
  `public int` **numVirtualPackageVertices**`( )`
  - **Usage**
    - Calculates the number of virtual package vertices
  - **Returns** - The number of virtual package vertices

B.1.5 CLASS **DependencyType**

An enumeration for the valid dependency types

DECLARATION

public final class DependencyType
extends java.lang.Enum

FIELDS

- public static final DependencyType RUN
  –
- public static final DependencyType INSTALL
  –
- public static final DependencyType CONFLICT
  –
- public static final DependencyType REPLACE
  –
- public static final DependencyType PROVIDES
  –

METHODS

- *getFromString*
  `public static DependencyType` **getFromString**`(`
  `java.lang.String` **typeString** `)`
  - **Usage**
    - Convert a string to the corresponding enumerated dependency type
  - **Parameters**
    - `typeString` - The string representing the dependency type
  - **Returns** - The corresponding dependency type

- *valueOf*
  `public static DependencyType` **valueOf**`( java.lang.String` **name** `)`

- *values*
  `public static final DependencyType` **values**`( )`

B.1.6 CLASS **DependencyVertex**

The base class for every vertex in a dependency graph

DECLARATION

public class DependencyVertex
extends DirectedSparseVertex

CONSTRUCTORS

- *DependencyVertex*
  `public` **DependencyVertex**`( )`

B.1.7 CLASS **Package**

The class modeling a Package

DECLARATION

public class Package
extends java.lang.Object

CONSTRUCTORS

- *Package*
  `public` **Package**`( java.lang.String` **unit**`, java.lang.String` **version** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `unit` - The unit name
    - `version` - The package version

- *Package*
  `public` **Package**`( java.lang.String` **unit**`, java.lang.String` **version**`, java.lang.String` **file** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `unit` - The unit name
    - `version` - The package version
    - `file` - The package file name

METHODS

- *addVirtualName*
  `public void` **addVirtualName**`( java.lang.String` **name** `)`
  - **Usage**
    - Add a virtual package name to the package
  - **Parameters**
    - `name` - The virtual package name
- *equals*
  `public boolean` **equals**`( java.lang.Object` **obj** `)`
- *getFile*
  `public String` **getFile**`( )`
- *getUnit*
  `public String` **getUnit**`( )`
- *getVersion*
  `public String` **getVersion**`( )`
- *getVirtualNames*
  `public Set` **getVirtualNames**`( )`
- *hashCode*
  `public int` **hashCode**`( )`
- *provides*
  `public boolean` **provides**`( java.lang.String` **virtualPackageName** `)`
  - **Usage**
    - Check if a package provides a virtual package name
  - **Parameters**
    - `virtualPackageName` - The virtual package name
  - **Returns** - true if virtualPackageName is provided, false otherwise
- *toString*
  `public String` **toString**`( )`

B.1.8 CLASS **PackageRepository**

This class models a package repository

DECLARATION

public class PackageRepository
extends java.util.HashSet

SERIALIZABLE FIELDS

- private Map packageNameMap
  –
- private Map virtualPackageNameMap
  –
- private DependencyGraph dependencyGraph
  –

CONSTRUCTORS

- *PackageRepository*
  `public` **PackageRepository**`( )`

METHODS

- *add*
  `public boolean` **add**`( org.edos_project.model.Package` **pkg** `)`
  - **Usage**
    - Add a package to the repository
  - **Parameters**
    - `pkg` - The package to add
  - **Returns** - true if the package has been succesfully added

- *addAlternativeDependency*
  `public void` **addAlternativeDependency**`(`
  `org.edos_project.model.util.PackageDependencyDataSet`
  pddSet `)`
  - **Usage**
    - Add an alternative dependency between packages
  - **Parameters**

    * `pddSet` - The dependency data bundle set

- *addDependency*
  `public void` **addDependency**`(`
  `org.edos_project.model.util.PackageDependencyData` **pdd** `)`
  - **Usage**
    * Add a dependency between packages
  - **Parameters**
    * `pdd` - The dependency data bundle

- *extract*
  `public PackageRepository` **extract**`( java.lang.String`
  unitName `)`
  - **Usage**
    * Extract the packages cone
  - **Parameters**
    * `unitName` - The unit name
  - **Returns** - A new package repository representing the extracted cone

---

- *findPackage*
  `public Package` **findPackage**`( java.lang.String` **unitName**`,`
  `java.lang.String` **packageVersion** `)`
  - **Usage**
    * Find a package
  - **Parameters**
    * `unitName` - The unit name
    * `packageVersion` - The version name
  - **Returns** - The found package or null if not found

- *findPackages*
  `public Set` **findPackages**`( java.lang.String` **unitName** `)`
  - **Usage**
    * Find all the packages with a given unit name
  - **Parameters**
    * `unitName` - The unit name
  - **Returns** - The set of found packages

- *findVirtualPackage*
  `public Set` **findVirtualPackage**`( java.lang.String`
  virtualPackageName `)`
  - **Usage**
    * Find all the packages that provide a given virtual package name
  - **Parameters**
    * `virtualPackageName` - The virtual package name
  - **Returns** - The set of found packages

- *getDependencyGraph*
  `public DependencyGraph` **getDependencyGraph**`( )`

- *getPackageAlternativeDependencyRelations*
  `public Set` **getPackageAlternativeDependencyRelations**`(`
  `org.edos_project.model.Package` **pkg**`,`
  `org.edos_project.model.DependencyType` **dt** `)`
  - **Usage**
    - Get all the package alternative dependencies of a given type
  - **Parameters**
    - `pkg` - The package
    - `dt` - The dependency type
  - **Returns** - The set of all the alternative dependency relationships for the package

---

- *getPackageDependencyRelations*
  `public Set` **getPackageDependencyRelations**`(`
  `org.edos_project.model.Package` **pkg**`,`
  `org.edos_project.model.DependencyType` **dt** `)`
  - **Usage**
    - Get all the package dependencies of a given type
  - **Parameters**
    - `pkg` - The package
    - `dt` - The dependency type
  - **Returns** - The set of all the dependency relationships for the package

---

- *getPurgedDependencyGraph*
  `public DependencyGraph` **getPurgedDependencyGraph**`( )`
  - **Usage**
    - Get the dependency graph without isolated vertices
  - **Returns** - The dependency graph without isolated vertices

---

- *getUnitNames*
  `public Set` **getUnitNames**`( )`
  - **Usage**
    - Get all the package unit names in the repository
  - **Returns** - The set of unit names

- *getVirtualPackageNames*
  `public Set` **getVirtualPackageNames**`( )`
  - **Usage**
    - Get all the virtual package names provided by the packages in the repository
  - **Returns** - The set of virtual package names

- *linkVirtualPackages*
  `public void` **linkVirtualPackages**`( )`
  - **Usage**
    - Add a link from each virtual package A to the standard package A if they both exist

B.1.9 CLASS StandardDependencyEdge

Models a standard dependency edge

DECLARATION

public class StandardDependencyEdge
extends org.edos_project.model.DependencyEdge

CONSTRUCTORS

- *StandardDependencyEdge*
  `public` **StandardDependencyEdge**`(`
  `org.edos_project.model.DependencyType` **dependencyType**`,`
  `org.edos_project.model.DependencyVertex` **source**`,`
  `org.edos_project.model.UnitVertex` **target**`,`
  `org.edos_project.model.VersionRelationship`
  versionRelationship `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `dependencyType` - The dependency type
    - `source` - The source vertex
    - `target` - The target unit vertex
    - `versionRelationship` - A version relation specifying the dependency data

METHODS

- *getVersionRelation*
  `public VersionRelationship` **getVersionRelation**`( )`
- *hashCode*
  `public int` **hashCode**`( )`
- *isAlternative*
  `public boolean` **isAlternative**`( )`
  - **Returns** - true if the dependency edge is an alternative edge
- *toString*
  `public String` **toString**`( )`

B.1.10 CLASS **StandardUnitVertex**

Model a standard unit vertex

DECLARATION

public class StandardUnitVertex
extends org.edos_project.model.UnitVertex

CONSTRUCTORS

- *StandardUnitVertex*
  `public` **StandardUnitVertex**`( java.lang.String` **unitName** `)`

METHODS

- *hashCode*
  `public int` **hashCode**`( )`
- *toString*
  `public String` **toString**`( )`

B.1.11 CLASS **UnitVertex**

Base class for all the vertices in the dependency graph

DECLARATION

public abstract class UnitVertex
extends org.edos_project.model.DependencyVertex

CONSTRUCTORS

- *UnitVertex*
  `public` **UnitVertex**`( java.lang.String` **name** `)`

METHODS

- *addPackage*
  `public void` **addPackage**`( org.edos_project.model.Package` **pkg** `)`
  - **Usage**

    * Associate a package to this vertex
  - **Parameters**
    * `pkg` - The package

- *getName*
  `public String` **getName**`( )`
  - **Returns** - The unit vertex name

- *getPackages*
  `public Set` **getPackages**`( )`
  - **Returns** - All the packages associated with this vertex

B.1.12 CLASS **VersionOperator**

An enumeration for the valid dependency operators

DECLARATION

public final class VersionOperator **extends** java.lang.Enum

FIELDS

- public static final VersionOperator LT
  –
- public static final VersionOperator LE
  –
- public static final VersionOperator EQ
  –
- public static final VersionOperator GE
  –
- public static final VersionOperator GT
  –
- public static final VersionOperator ALL
  –

METHODS

- *getFromString*
  `public static VersionOperator` **getFromString**`(`
  `java.lang.String` **operatorString** `)`
  - **Usage**
    - Convert a string to the corresponding enumerated operator type
  - **Parameters**
    - `operatorString` - The string representing the operator type
  - **Returns** - The corresponding operator type
- *valueOf*
  `public static VersionOperator` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final VersionOperator` **values**`( )`

B.1.13 CLASS VersionRelationship

A class modeling a relationship between packages

DECLARATION

public class VersionRelationship **extends** java.lang.Object

CONSTRUCTORS

- *VersionRelationship*
  `public` **VersionRelationship**`( org.edos_project.model.Package` **sourcePackage**`, org.edos_project.model.VersionOperator` **operator**`, java.lang.String` **targetPackageUnit**`,`
  `java.lang.String` **targetPackageVersion** `)`
  - **Usage**
    - Constructor
  - **Parameters**
    - `sourcePackage` - The source package
    - `operator` - The relationship operator
    - `targetPackageUnit` - The target package unit
    - `targetPackageVersion` - The target package name

METHODS

- *equals*
  `public boolean` **equals**`( java.lang.Object` **obj** `)`
- *getOperator*
  `public VersionOperator` **getOperator**`( )`
- *getSourcePackage*
  `public Package` **getSourcePackage**`( )`
- *getTargetPackageUnit*
  `public String` **getTargetPackageUnit**`( )`
- *getTargetPackageVersion*
  `public String` **getTargetPackageVersion**`( )`
- *hashCode*
  `public int` **hashCode**`( )`
- *toString*
  `public String` **toString**`( )`

B.1.14 CLASS **VirtualPackageEdge**

This class models a provides relationship and links a standard unit vertex to the virtual package vertex whose name is the name of a virtual package provided by some package associated with the unit vertex.

DECLARATION

public class VirtualPackageEdge
extends org.edos_project.model.DependencyEdge

CONSTRUCTORS

- *VirtualPackageEdge*
  `public` **VirtualPackageEdge**`(`
  `org.edos_project.model.VirtualPackageVertex` **source**`,`
  `org.edos_project.model.StandardUnitVertex` **target** `)`

B.1.15 CLASS **VirtualPackageVertex**

This class models a virtual package vertex

DECLARATION

public class VirtualPackageVertex
extends org.edos_project.model.UnitVertex

CONSTRUCTORS

- *VirtualPackageVertex*
  `public` **VirtualPackageVertex**`( java.lang.String` **name** `)`

METHODS

- *toString*
  `public String` **toString**`( )`

Appendix C

Package org.edos_project.io

Package Contents *Page*

Classes

This class encapsulates all the behavior for handling EGraph files.

C.1 Classes

C.1.1 CLASS **EGraph**

This class encapsulates all the behavior for handling EGraph files.

DECLARATION

public class EGraph
extends java.lang.Object

CONSTRUCTORS

- *EGraph*
  `public` **EGraph**`( )`

METHODS

- *load*
  `public static PackageRepository` **load**`( java.io.File` **file** `)`
  - **Usage**
    - Load a package repository from EGraph file
  - **Parameters**
    - `file` - The file to load the package repository from
  - **Returns** - The loaded package repository
  - **Exceptions**
    - `javax.xml.parsers.ParserConfigurationException` -
    - `org.xml.sax.SAXException` -
    - `java.io.IOException` -

- *load*
  `public static PackageRepository` **load**`( java.lang.String` **fileName** `)`
  - **Usage**
    - Load a package repository from EGraph file
  - **Parameters**
    - `fileName` - The file name to load the package repository from
  - **Returns** - The loaded package repository
  - **Exceptions**
    - `javax.xml.parsers.ParserConfigurationException` -
    - `org.xml.sax.SAXException` -
    - `java.io.IOException` -

- *save*
  `public static void` **save**`(`
  `org.edos_project.model.PackageRepository`
  packageRepository`, java.io.File` **file** `)`
  - **Usage**
    - Save a package repository to an EGraph file
  - **Parameters**
    - `packageRepository` - The package repository to save
    - `file` - The EGraph file
  - **Exceptions**
    - `java.io.IOException` -

- *save*
  `public static void` **save**`(`
  `org.edos_project.model.PackageRepository`
  packageRepository`, java.lang.String` **fileName** `)`
  - **Usage**
    - Save a package repository to an EGraph file
  - **Parameters**
    - `packageRepository` - The package repository to save
    - `fileName` - The EGraph file name
  - **Exceptions**
    - `java.io.IOException` -

- *saveNew*
  `public static void` **saveNew**`(`
  `org.edos_project.model.PackageRepository`
  packageRepository`, java.io.File` **file** `)`
  - **Usage**
    - Save a package repository to an new XML file format
  - **Parameters**
    - `packageRepository` - The package repository to save
    - `file` - The output file
  - **Exceptions**
    - `java.io.IOException` -

- *saveNew*
  `public static void` **saveNew**`(`
  `org.edos_project.model.PackageRepository`
  packageRepository`, java.lang.String` **fileName** `)`
  - **Usage**
    - Save a package repository to an new XML file format
  - **Parameters**
    - `packageRepository` - The package repository to save
    - `fileName` - The output file name
  - **Exceptions**
    - `java.io.IOException` -

Appendix D

Contribution to the Workshop on Future Research Challenges for Software and Services (FRCSS06)

The following is the contribution of the WorkPackage2 Team to the Workshop on Future Research Challenges for Software and Services (FRCSS06)

Maintaining large software distributions: new challenges from the FOSS era*

Roberto Di Cosmo[1], Berke Durak[2], Xavier Leroy[2], Fabio Mancinelli[1], and Jérôme Vouillon[1]

[1] PPS, University of Paris 7, Firstname.Lastname@pps.jussieu.fr
[2] INRIA Rocquencourt, Firstname.Lastname@inria.fr

Abstract. In the mainstream adoption of free and open source software (FOSS), *distribution editors* play a crucial role: they package, integrate and distribute a wide variety of software, written in a variety of languages, for a variety of purposes of unprecedented breadth.
Ensuring the quality of a FOSS distribution is a technical and engineering challenge, owing to the size and complexity of these distributions (tens of thousands of software packages). A number of original topics for research arise from this challenge. This paper is a gentle introduction to this new research area, and strives to clearly and formally identify many of the desirable properties that must be enjoyed by these distributions to ensure an acceptable quality level.

1 Introduction

Managing large software systems has always been a stimulating challenge for the research field in Computer Science known as Software Engineering. Many seminal advances by founding fathers of Comp. Sci. were prompted by this challenge (see the book "Software Pioneers", edited by M. Broy and E. Denert [1], for an overview). Concepts such as structured programming, abstract data types, modularization, object orientation, design patterns or modeling languages (unified or not) [2, 3], were all introduced with the clear objective of simplifying the task not only of the programmer, but of the software engineer as well.

Nevertheless, in the recent years, two related phenomena: the explosion of Internet connectivity and the mainstream adoption of free and open source software (FOSS), have deeply changed the scenarii that today's software engineers face. The traditional organized, safe world where software is developed from specifications in a fully centralized way is no longer the only game in town. We see more and more complex software systems that are assembled from loosely coupled sources developed by programming teams not belonging to any single company, cooperating only through fast Internet connections. The availability of code distributed under FOSS licences makes it possible to reuse such code

* This work was supported by the EDOS Specific Targeted Research Project of the 6th European Union Framework Programme.

without formal agreements among companies, and without any form of central authority that coordinates this burgeoning activity.

This has led to the appearance of the so-called *distribution editors*, who try to offer some kind of reference viewpoint over the breathtaking variety of FOSS software available today: they take care of packaging, integrating and distributing tens of thousands of software packages, very few being developed in-house and almost all coming from independent developers. We believe that the role of distribution editors is deeply novel: no comparable task can be found in the traditional software development and distribution model.

This unique position of a FOSS distribution editor means that many of the standard, often unstated assumptions made for other complex software systems no longer hold: there is no common programming language, no common object model, no common component model, no central authority, neither technical nor commercial[3].

Consequently, most FOSS distribution today simply rely on the general notion of software *package*[4]: a bundle of files containing data, programs, and configuration information, with some metadata attached. Most of the metadata information deals with *dependencies*: the relationships with other packages that may be needed in order to run or install a given package, or that conflict with its presence on the system.

We now give a general description of a typical FOSS process. In figure 1 we have an imaginary project, called `foo`, handled by two developers, Alice Torvalds and Bob Dupont, who use a common CVS or Subversion repository and associated facilities such as mailing lists at a typical FOSS development site such as Sourceforge. Open source software is indeed developed as *projects*, which may group one or more developers. Projects can be characterized by a common goal and the use of a common infrastructure, such as a common version control repository, bug tracking system, or mailing lists. For instance, the Firefox browser, the Linux kernel, the KDE and Gnome desktop environments or the GNU C compiler are amongst the largest FOSS projects and have their own infrastructures. Of course, even small bits of software like `sysstat` consitute projects, even if they are developed by only one author without the use of a version control system. A given project may lead to one or more *products*. For instance, the KDE project leads to many products, from the `konqueror` browser to the desktop environment itself. Each FOSS product may then be included in a distribution. In our example, the project `foo` delivers the products `gfoo`, `kfoo` and `foo-utils`. A *port* is the inclusion of a product into a distribution by one or more *maintainers* of that distribution. The maintainers must:

– Import and regularly track the source code for the project into the distribution's own version control or storage system (this is depicted in figure 1 by a

[3] In the world of Windows-based personal computing, for example, the company controlling Windows can actually impose to the ISV the usage of its API and other rules.

[4] Not to be mistaken for the software organizational unit present in many modern programming languages.

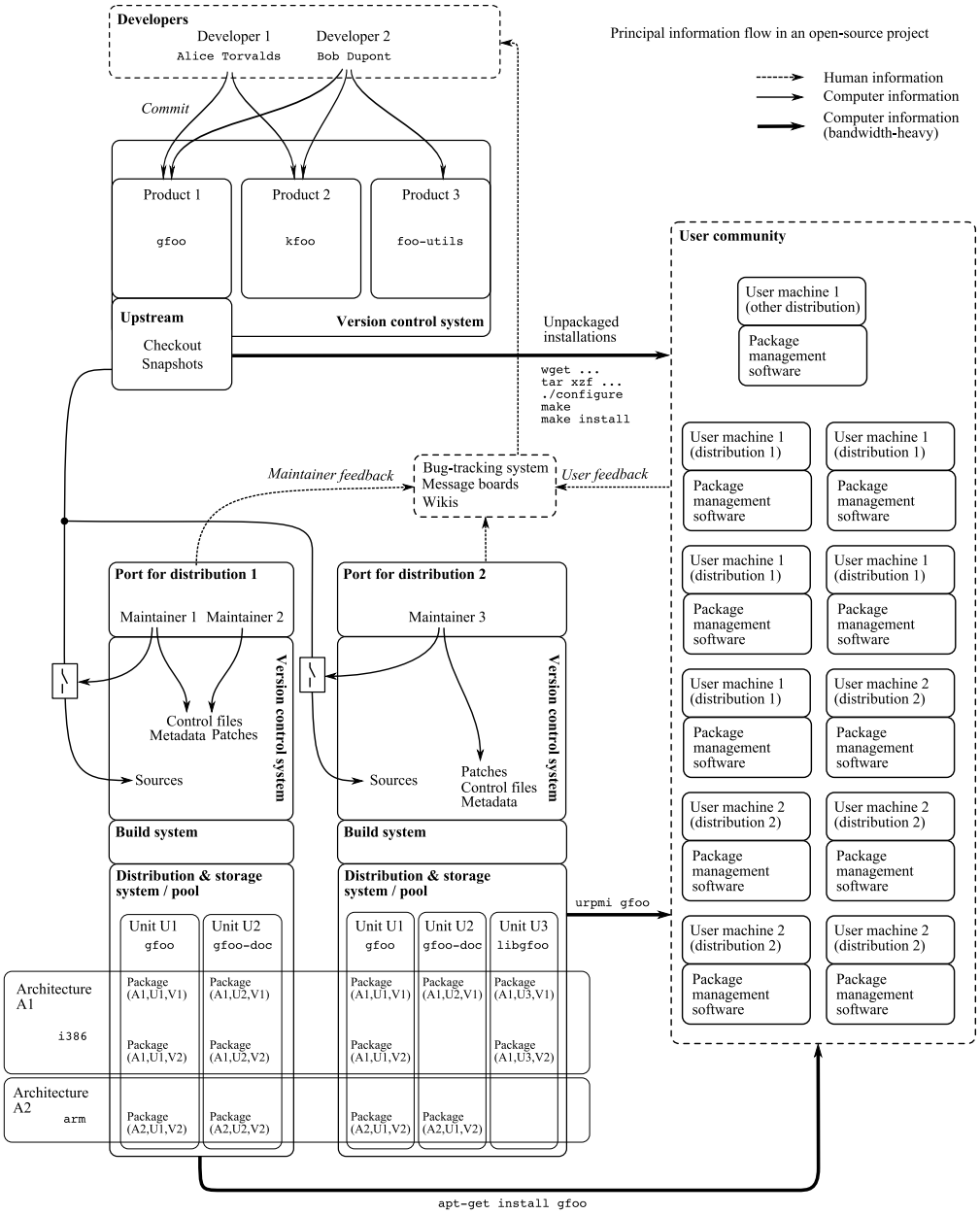


Fig. 1. Major flow of information in a FOSS project.

switch controlling the flow of information from the upstream to the version control system of the distribution).
- Ensure that the dependencies of the product are already included in the distribution.
- Write or include patches to adapt the program to the distribution.
- Write installation, upgrading, configuration and removal scripts.
- Write metadata and control files.
- Communicate with the upstream developers by forwarding them bug reports, patches or feature requests.

We see that the job of maintainers is substantial for which attempts to automate some of those tasks, such as automated dependency extraction tools [4, 5] or getting source code updates from developers [6] are no substitute. In our example, we have a Debian-based distribution 1, with two maintainers for `foo`, and an RPM-based distribution 2 with one maintainer. A given product will be divided into one or more *units*, which will be compiled for the different *architectures* supported by the distribution (a given unit may not be available on all architectures) and bundled as *packages*. The metadata and control files specify how the product is divided into units, how each unit is to be compiled and packaged and on which architectures, as well as the dependency information, the textual description of the units, their importance, and classification tags. These packages are then automatically downloaded (as well as their dependencies) by the package management software (for instance, `apt` or `urpmi`) of the users of that distribution. Some users may prefer to download directly the sources from the developers, in which case they will typically execute a sequence of commands such as `./configure && make && make install` to compile and install that software. However, they then lose the many benefits of a package management system, such as tracking of the files installed by the package, automated installation of the dependencies, local modifications and installation scripts.

We now turn to the problem of ensuring the quality of a distribution. This problem is the focus of the European FP6 project EDOS (Environment for the development and Distribution of Open Source software). This problem can therefore be divided into three main tasks:

Upstream tracking makes sure that the package in the distribution closely follows the evolution of the software development, almost always carried over by some team outside the control of the distributor.

Testing and integration makes sure that the program performs as expected in combination with other packages in the distribution. If not, bug reports need propagating to the upstream developer.

Dependency management makes sure that, in a distribution, packages can be installed and user installations can be upgraded when new versions of packages are produced, while respecting the constraints imposed by the dependency metadata.

In this paper, we focus on the last task: dependency management. This task is surprisingly complex [4, 7], owing to the large number of packages present in a

typical distribution and to the complexity and richness of their interdependencies. It is at the very heart of the research activity conducted in workpackage 2 of the EDOS project.

More specifically, our focus is on the issues related to dependency management for large sets of software packages, with a particular attention to what must be done to maintain consistency of a software distribution *on the repository side*, as opposed to maintaining a set of packages installed *on a client machine*.

This choice is justified by the following observation: maintaining consistency of a distribution of software packages is *fundamental* to ensure quality and scalability of current and future distributions; yet, it is also an *invisible* task, since the smooth working it ensures on the end user side tends to be considered as normal and obvious as the smooth working of packet routing on the Internet. In other words, we are tackling an essential *infrastructure* problem that has long been ignored: while there are a wealth of client-side tools to maintain a user installation (`apt`, `urpmi`, `smart` and many others [8–10]), there is surprisingly little literature and publically available tools that address server-side requirements. We found very little significant prior work in this area, despite it being critical to the success of FOSS in the long term.

The paper is organised as follows. Section 2 contains a formal description of the main characteristics of a software package found in the mainstream FOSS distributions, as far as dependency are concerned. In Section 3 we identify and formally define three desirable properties of a distribution with respect to dependency management. Section 4 discusses the feasibility of checking these properties. A few empirical measurements are given in section 5, followed by conclusions in section 6.

2 Basic definitions

Every package management system [11, 12] takes into account the interrelationships among packages (to different extents). We will call these relationships *requirements*. Several kinds of requirements can be considered. The most common one is a *dependency* requirement: in order to install package P_1, it is necessary that package P_2 is installed as well. Less often, we find *conflict* requirements: package P_1 cannot coexist with package P_2.

Some package management systems specialize these basic types of requirements by allowing to specify the *timeframe* during which the requirement must be satisfied. For example, it is customary to be able to express *pre-dependencies*, a kind of dependency stating that a package P_1 needs package P_2 to be present on the system *before* P_1 can be installed [11].

In the following, we assume the distribution and the architecture are fixed. We will identify packages, which are archive files containing metadata and installation scripts, with pairs of a unit and a version.

Definition 1 (Package, unit). *A* package *is a pair* (u, v) *where* u *is a unit and* v *is a version. Units are arbitrary strings, and we assume that versions are non-negative integers.*

While the ordering over version strings as used in common OSS distributions is not discrete (i.e., for any two version strings v_1 and v_2 such that $v_1 < v_2$, there exists v_3 such that $v_1 < v_3 < v_2$), taking integers as version numbers is justified for two reasons. First, any given repository will have a finite number of packages. Second, only packages with the same unit will be compared.

For instance, if our Debian repository contains the versions `2.15-6`, `2.16.1-cvs20051117-1` and `2.16.1cvs20051206-1` of the unit `binutils`, we may encode these versions respectively as 0,1 and 2, giving the packages (`binutils`, 0), (`binutils`, 1), and (`binutils`, 2).

Definition 2 (Repository). *A* repository *is a tuple $R = (P, D, C)$ where P is a set of packages, $D : P \to \mathscr{P}(\mathscr{P}(P))$ is the dependency function*[5]*, and $C \subseteq P \times P$ is the conflict relation. The repository must satisfy the following conditions:*

- *The relation C is symmetric, i.e., $(\pi_1, \pi_2) \in C$ if and only if $(\pi_2, \pi_1) \in C$ for all $\pi_1, \pi_2 \in P$.*
- *Two packages with the same unit but different versions conflict*[6]*, that is, if $\pi_1 = (u, v_1)$ and $\pi_2 = (u, v_2)$ with $v_1 \neq v_2$, then $(\pi_1, \pi_2) \in C$.*

In a repository $R = (P, D, C)$, the dependencies of each package p are given by $D(p) = \{d_1, \ldots, d_k\}$ which is a set of sets of packages, interpreted as follows. If p is to be installed, then all its k dependencies must be satisfied. For d_i to be satisfied, at least one of the packages of d_i must be available. In particular, if one of the d_i is the empty set, it will never be satisfied, and the package p is not installable.

Example 1. Let $R = (P, D, C)$ be the repository given by

$$
\begin{aligned}
&P = \{a, b, c, d, e, f, g, h, i, j\} \\
&D(a) = \big\{\{b\}, \{c, d\}, \{d, e\}, \{d, f\}\big\} \\
&D(b) = \big\{\{g\}\big\} \quad D(c) = \big\{\{g, h, i\}\big\} \quad D(d) = \big\{\{h, i\}\big\} \\
&D(e) = D(f) = \big\{\{j\}\big\} \\
&C = \{(c, e), (e, c), (e, i), (i, e), (g, h), (h, g)\}
\end{aligned}
$$

where $a = (u_a, 0)$, $b = (u_b, 0)$, $c = (u_c, 0)$ and so on. The repository R is represented in figure 2. For the package a to be installed, the following packages must be installed: b, either c or d, either d or e, and either d or f. Packages c and e, e and i, and g and h cannot be installed at the same time.

In computer science, dependencies are usually *conjunctive*, that is they are of the form

$$a \to b_1 \wedge b_2 \wedge \cdots \wedge b_s$$

[5] We write $\mathscr{P}(X)$ for the set of subsets of X.

[6] This requirement is present in some package management systems, notably Debian's, but not all. For instance, RPM-based distributions allow simultaneous installation of several versions of the same unit, at least in principle.

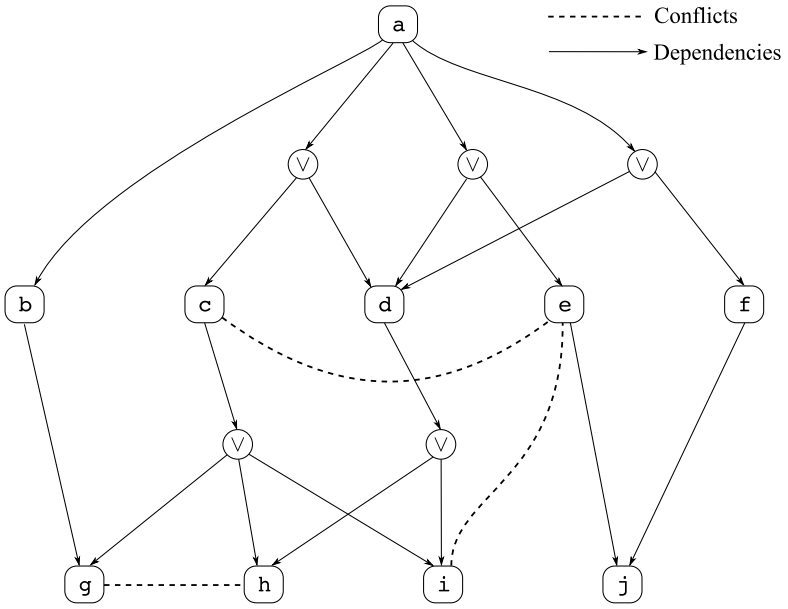


Fig. 2. The repository of example 1.

where a is the target and b_1, b_2, ... are its prerequisites. This is the case in `make` files, where all the dependencies of a target must be built before building the target. Such dependency information can be represented by a directed graph, and dependencies can be solved by the well-known topological sort algorithm. Our dependencies are of a more complex kind, which we name *disjunctive* dependencies. Their general form is a conjunction of disjunctions:

$$a \rightarrow (b_1^1 \vee \cdots \vee b_1^{r_1}) \wedge \cdots \wedge (b_s^1 \vee \cdots \vee b_s^{r_s}). \tag{1}$$

For a to be installed, each term of the right-hand side of the implication 1 must be satisfied. In turn, the term $b_i^1 \vee \cdots \vee b_i^{r_i}$ when $1 \leq i \leq s$ is satisfied when at least one of the b_i^j with $1 \leq j \leq r_i$ is satisfied. If a is a package in our repository, we therefore have

$$D(a) = \{\{b_1^1, \ldots, b_1^{r_1}\}, \cdots, \{b_s^1, \ldots, b_s^{r_s}\}\}.$$

In particular, if one of the terms is empty (if $\varnothing \in D(a)$), then a cannot be satisfied. This side-effect is useful for modeling repositories containing packages mentioning another package b that is not in that repository. Such a situation may occur because of an error in the metadata, because the package b has been removed, or b is in another repository, maybe for licensing reasons.

Concerning the relation C, two packages $\pi_1 = (u_1, v_1), \pi_2 = (u_2, v_2) \in P$ conflict when $(\pi_1, \pi_2) \in C$. Since conflicts are a function of presence and not of installation order, the relation C is symmetric.

Definition 3 (Installation). *An* installation *of a repository $R = (P, D, C)$ is a subset of P, giving the set of packages installed on a system. An installation is* healthy *when the following conditions hold:*

- ***Abundance:*** *Every package has what it needs. Formally, for every $\pi \in I$, and for every dependency $d \in D(\pi)$ we have $I \cap d \neq \varnothing$.*
- ***Peace:*** *No two packages conflict. Formally, $(I \times I) \cap C = \varnothing$.*

Definition 4 (Installability and co-installability). *A package π of a repository R is* installable *if there exists a healthy installation I such that $\pi \in I$. Similarly, a set of packages Π of R is* co-installable *if there exists a healthy installation I such that $\Pi \subseteq I$.*

Note that because of conflicts, every member of a set $X \subseteq P$ may be installable without the set X being co-installable.

Example 2. Assume a depends on b, c depends on d, and c and d conflict. Then, the set $\{a, b\}$ is not co-installable, despite each of a and b being installable and not conflicting directly.

Definition 5 (Maximal co-installability). *A set X of co-installable packages of a repository R is* maximal *if there is no other co-installable subset X' of R that strictly contains X. We write* $\mathrm{maxco}(R)$ *for the family of all maximal co-installable subsets of R.*

Definition 6 (Dependency closure). *The* dependency closure $\Delta(\Pi)$ *of a set of package Π of a repository R is the smallest set of packages included in R that contains Π and is closed under the* immediate dependency *function $\overline{D} : \mathscr{P}(P) \to \mathscr{P}(P)$ defined as*

$$\overline{D}(\Pi) = \bigcup_{\substack{\pi \in \Pi \\ d \in D(\pi)}} d.$$

In simpler words, $\Delta(\Pi)$ contains Π, then all packages that appear as immediate dependencies of Π, then all packages that appear as immediate dependencies of immediate dependencies of Π, and so on. Since the domain of $\overline{D}$ is a complete lattice, and $\overline{D}$ is clearly a continuous function, we immediately get (by Tarski's theorem) that such a smallest set exists and can be actually computed as follows:

Proposition 1. *The dependency closure $\Delta(\Pi)$ of Π is:*

$$\Delta(\Pi) = \bigcup_{n \geq 0} \overline{D}^n(\Pi).$$

The notion of dependency closure is useful to extract the part of a repository that pertains to a package or to a set of packages.

Definition 7 (Generated subrepository). *Let $R = (P, D, C)$ be a repository and $\Pi \subseteq P$ be a set of packages. The* subrepository generated by *Π is the repository $R|_\Pi = (P', D', C')$ whose set of packages is the dependency closure of Π and whose dependency and conflict relations are those of R restricted to that set of packages. More formally we have $P' = \Delta(\Pi)$, $D' : P' \to \mathscr{P}(\mathscr{P}(P')), \pi \mapsto \{d \cap P' \mid d \in D(\pi)\}$ and $C' = C \cap (P' \times P')$.*

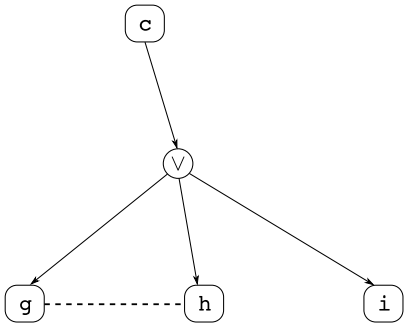


Fig. 3. The subrepository generated by package c. The dependency closure is $\{c, g, h, i\}$.

Figure 3 shows the subrepository generated by the package c of example 1. The dependency closure of c is the set of package nodes of that subrepository. A larger, real-world example is shown in figure 4.

We then have the following property, which allows to consider only the relevant subrepositories when answering questions of installability.

Proposition 2 (Completeness of subrepositories). *A package π is installable w.r.t. R if and only if it is installable w.r.t. $R|_\pi$. (Similarly for co-installability.)*

3 Maintaining a package repository

The task of maintaining a package repository is difficult: the maintainance team must monitor the evolution of thousand of packages over time, and address the error reports coming from different sources (users, QA teams, developers, etc.). It is desirable to automate as much of this work as possible. Our medium-term goal is to build tools that help distribution maintainers track dependency-related problems in package repositories. We detail here some of the desirable properties of a repository. The first is *history-free*, in that it applies to a given state of a repository.

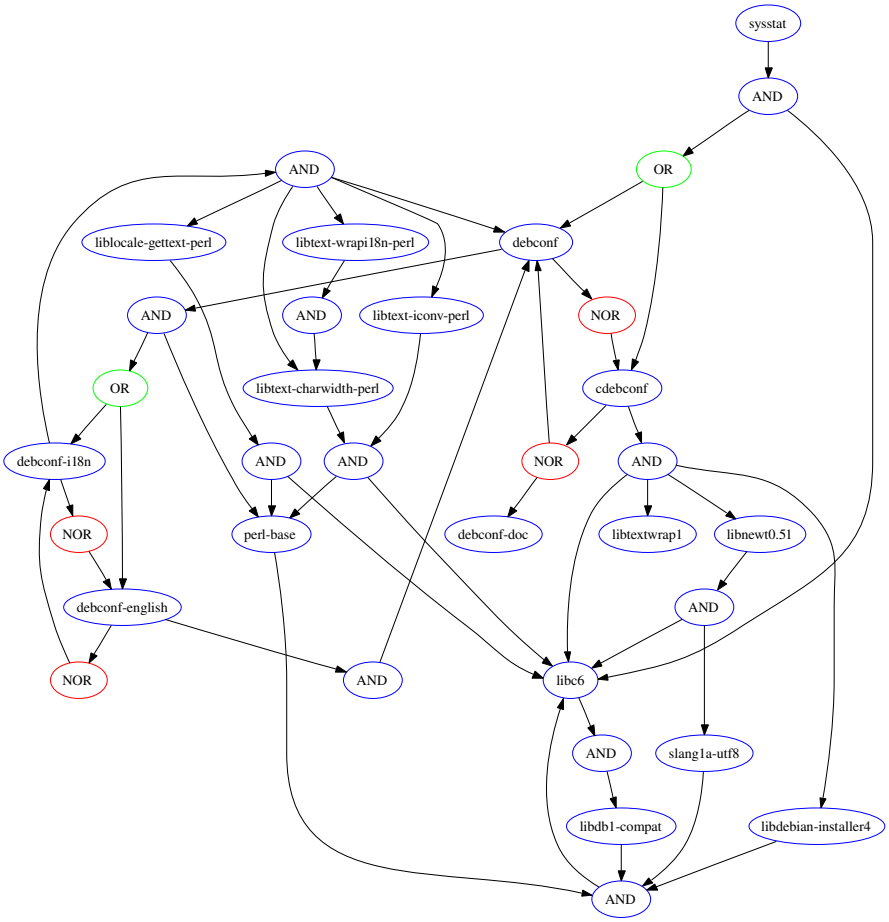


Fig. 4. The subrepository generated by the package `sysstat` in Debian stable on 2005-12-13, viewed as a boolean graph. Version numbers have been omitted for brevity.

Being trimmed We say that a repository R is *trimmed* when every package of R is installable w.r.t. R. The intuition behind this terminology is that a non-trimmed repository contains packages that cannot be installed in any configuration. We call those packages *broken*. They behave as if they were not part of the repository. It is obviously desirable that at any point in time, a repository is trimmed, that is, contains no broken packages.

The next properties are *history-sensitive*, meaning that they take into account the evolution of the repository over time. Due to this dependency on time, the precise formulation of these properties is delicate. Just like history-free properties are relevant to users who install a distribution from scratch, history-sensitive properties are relevant to users who upgrade an existing installation.

Monotonicity Let R_t be the repository at time t and consider a coinstallable set of packages C_t. Some users can actually have packages C_t installed simultaneously on their system. These users have the possibility of installing additional packages from R_t, resulting in a coinstallable set of packages C'_t. These users can reasonably expect that they will be able to do so (extend C_t into C'_t) at any future time t', using the repository $R_{t'}$, which, being *newer*, is supposed to be *better* than the old R_t.

Of course, users are ready to accept that in $R_{t'}$ they will not get exactly ${C'}_t$, but possibly ${C'}_{t'}$, where some packages were updated to a greater version, and some others have been replaced as the result of splitting into smaller packages or grouping into larger ones. But, clearly, it is not acceptable to evolve R_t into $R_{t'}$ if R_t allows to install, say, `apache` together with `squid`, while $R_{t'}$ does not.

We say that a repository history line is *monotone* if the *freedom* of a user to install packages is a monotone function of time. Writing $F(x, R)$ for the set of possible package sets in R that are a possible replacement of package x according to the metadata, monotonicity can be formally expressed as

$$\mathrm{Mon}(R) = \forall t < t'.\ \forall P \in \mathrm{Con}(R_t).\ \exists Q \in \mathrm{Con}(R_{t'}).\ \forall x \in P.\ Q \cap F(x, R_{t'}) \neq \varnothing$$

Upgradeability Another reasonable expectation of the user is to be able to upgrade a previously installed package to the most recent version (or even any more recent version) of this package that was added to the repository since her latest installation. She is ready to accept that this upgrade will force the installation of some new packages, the upgrade of some other packages, and the replacement of some sets of package by other sets of packages, as the result of the reorganization of the structure of the packages. However, she cannot accept that the upgrade of a package forces the complete removal of other previously installed packages that she uses.

In other terms, the evolution of a repository respects the upgradeability property if all upgrades of individual packages can be performed without loss of functionality: all packages removed as part of such an upgrade must be compensated by the installation of other packages of equivalent functionality. The notion of "equivalent functionality" needs to be indicated in the metadata of the packages, such as for instance the "Replaces" clauses in Debian's package metadata.

We remark that these properties are *not* interdefinable. We give here a proof of this assertion by exhibiting example repositories showing this independence of the properties. For the first two cases, consider three repositories R_1, R_2, R_3 whose sets of packages are $P_1 = \{(a,1),(b,1),(c,1)\}$, $P_2 = \{(a,1),(b,1)\}$, $P_3 = \{(a,1),(a,2),(b,1)\}$ with no conflicts nor dependencies among the version 1 packages and a conflict among $(a,2)$ and $(b,1)$. Notice that at each moment t in time, R_t is trimmed.

1. A repository that stays trimmed over a period of time is not necessarily monotone, nor upgradeable. Since $(c,1)$ disappears between times 1 and 2, this step in the evolution does not preserve monotonicity. Since $(a,2)$ has a new conflict (namely with $(b,1)$) in R_3, the evolution from R_2 to R_3 does not preserve upgradeability.
2. A repository that stays trimmed over a period of time and evolves in a monotone fashion is not necessarily upgradeable. The evolution from R_2 to R_3 above is monotone, each of R_2 and R_3 is trimmed, but we fail upgradeability because there is no way of going from $\{(a,1),(b,1)\}$ to $\{(a,2),(b,1)\}$ because of the conflict.
3. A repository that stays trimmed over a period of time and is upgradeable is not necessarily monotone.
   Consider repositories R_1 and R_2 with $P_1 = \{(a,1),(b,1)\}$ and $P_2 = \{(a,2),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are isolated packages, while $(a,2)$ conflicts with $(b,1)$. Now, a user having installed all of R_1 and really willing to get $(a,2)$ can do it, but at the price of giving up $(b,1)$. This evolution of the repository is therefore upgradeable but not monotone.
4. A repository that evolves in a monotone and upgradeable fashion is not necessarily trimmed at any time: indeed, the monotonicity and upgradeability property only speak of *consistent* subsets of a repository, that cannot contain, by definition, any broken packages.
   Consider for example repositories R_1, R_2 with $P_1 = \{(a,1)\}$, $P_2 = \{(a,1),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are broken because they depend on a missing package $(c,1)$. Here, the evolution of R_1 to R_2 is trivially monotone and upgradeable, because there is *no* consistent subset of R_1 and R_2, and both R_1 and R_2 are not trimmed because they contain broken packages.

The examples above to prove that the three properties are actually independent may seem contrived, but are simplifications of real-world scenarii. For instance, example 3 can actually happen in the evolution of real repositories, when for some reason the new version of a set of interrelated packages is only partially migrated to the repository. Many packages are split into several packages to isolate architecture-independent files, as in the Debian packages `swi-prolog` and `swi-prolog-doc`. When performing this split, it is quite natural to add a conflict in `swi-prolog-doc` against old, non-splitted versions of `swi-prolog`. If the new version of `swi-prolog-doc` slips into a real repository before the new, splitted version of `swi-prolog`, we are exactly in situation number 3 above.

Package developers seem aware of some of these issues: they actually do their best to ensure monotonicity and upgradeability by trying to reduce as much as possible the usage of conflicts, and sometime resorting to naming conventions for the packages when a radical change in the package happens, like in the case of `xserver-common` vs. `xserver-common-v3` in Debian, as can be seen in the dependencies for `xserver-common`.

```
Package: xserver-common
Conflicts: xbase (<< 3.3.2.3a-2), xsun-utils, xbase-clients (<< 3.3.6-1),
  suidmanager (<< 0.50), configlet (<= 0.9.22),
  xserver-3dlabs (<< 3.3.6-35), xserver-8514 (<< 3.3.6-35),
  xserver-agx (<< 3.3.6-35), xserver-common-v3 (<< 3.3.6-35),
  xserver-fbdev (<< 3.3.6-35), xserver-i128 (<< 3.3.6-35),
  xserver-mach32 (<< 3.3.6-35), xserver-mach64 (<< 3.3.6-35),
  xserver-mach8 (<< 3.3.6-35), xserver-mono (<< 3.3.6-35),
  xserver-p9000 (<< 3.3.6-35), xserver-s3(<< 3.3.6-35),
  xserver-s3v (<< 3.3.6-35), xserver-svga (<< 3.3.6-35),
  xserver-tga (<< 3.3.6-35), xserver-vga16 (<< 3.3.6-35),
  xserver-w32 (<< 3.3.6-35), xserver-xsun (<< 3.3.6-35),
  xserver-xsun-mono (<< 3.3.6-35), xserver-xsun24 (<< 3.3.6-35),
  xserver-rage128, xserver-sis
```

4 Algorithmic considerations

Our research objective within the EDOS project is to formally define the desirable properties of repositories stated in section 3 (and possibly other properties that will appear useful), and to develop efficient algorithms to check these properties automatically.

It is really not evident that any of these problems are actually tractable in practice: due to the rich language allowed to describe package dependencies in the mainstream FOSS distributions, even the simplest problems (checking installability of a single package) may involve verifications over a large number of other packages. During our first investigations of these problems, we have indeed already proven the following complexity result.

Theorem 1 (Package installability is an NP-complete problem). *Checking whether a single package P can be installed, given a repository R, is NP-complete.*

The full proof of this result will be published separately. It relies on a simple, polynomial-time reduction of the 3SAT problem to the installability problem. Given an instance of 3SAT, a repository is constructed having one package for the whole 3SAT formula, one package per clause of that formula, and three packages for each propositional atom occurring in that formula. Dependencies and conflicts between these packages are added in such a way that the package for the whole formula is installable if and only if the 3SAT formula is satisfiable.

Nevertheless, this strong limiting result does not mean that we will not be able to decide installability and the other problems in practice: the actual instances of these problems, as found in real repositories, could be quite simple in the average.

In particular, the converse of the reduction used for the NP-completeness proof leads to an effective way of deciding package installability. We developed an algorithm that encodes a repository R and its dependencies as a Boolean formula $C(R)$. (Details of the encoding will be published in a forthcoming paper.) Assignments of truth values to boolean variables that satisfy $C(R)$ are in one-to-one correspondence with sets of co-installable packages. Therefore, a package P is installable if and only if the Boolean formula $C(R) \wedge P$ is satisfiable, which we can check relatively efficiently using off-the-shelf SAT solving technology.

We implemented the conversion algorithm as well a SAT solver [13] and ran it over both the Debian pool (over 30,000 packages) and the Mandriva Cooker distribution (around 5,000 packages). The execution time is entirely acceptable, and the tool found a number of non-installable packages in both distributions.

We are now focusing our attention on the two time-dependent desirable properties for the repositories, which are, algorithmically speaking, much harder.

5 Empirical measurements

In parallel with our formal complexity and algorithmic investigations, we also performed some empirical measurements on the Debian and Mandriva distributions, to try and grasp the practical complexity of the problems.

Figure 5 gives a histogram showing the number of packages as a function of the size of the dependency closure, from the Debian stable, unstable and testing pools on 2005-12-13, which has 31149 packages. The average closure size is 158; 50% of the packages have a closure size of 71 or less, 90% of 372 or less, and 99% of 1077 or less. These numbers show that naive combinatorial algorithms, exponential in the size of the dependency closure, are clearly out of the question.

Figure 6 estimates the complexity of solving the Boolean formulae generated by our encoding of the installability problem. The "temperature" T of a formula in 3SAT conjunctive normal form is defined as $T = m/n$ where m is the number of clauses and n the number of variables. There is strong theoretical and practical evidence that hard SAT problems have a temperature close to 4.2, while SAT problems with temperatures well below or above that limit are easier to solve. The temperatures for the SAT problems corresponding to installability of the Debian packages range from 0.75 to 1.49, well below the threshold value of 4.2. This result confirms that we are dealing with relatively easy satisfiability problems, maybe owing to the small-world nature of the dependency graphs [14].

6 Conclusions

We have presented and motivated in this paper three fundamental properties for large repositories of FOSS packages that are quite different from the usual

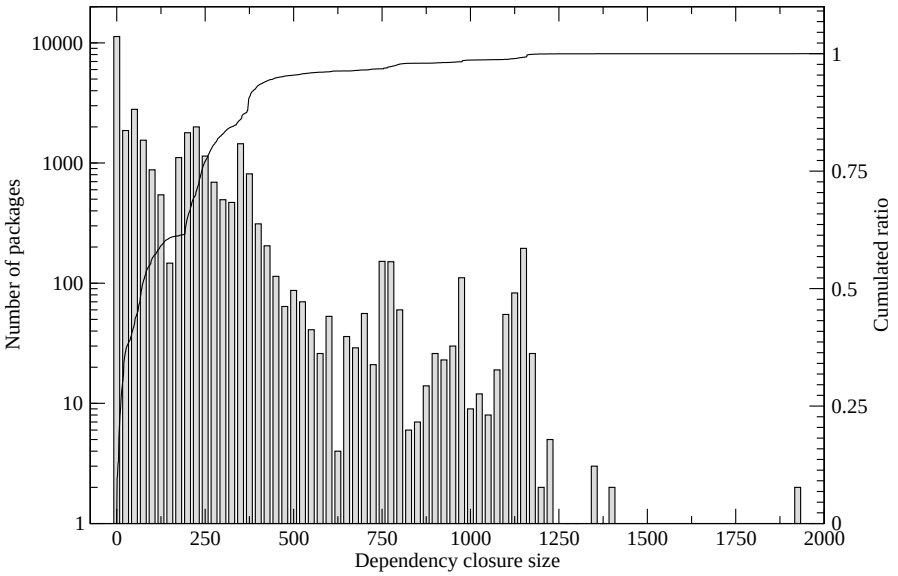


Fig. 5. Number of packages as a function of the size of their dependency closures.

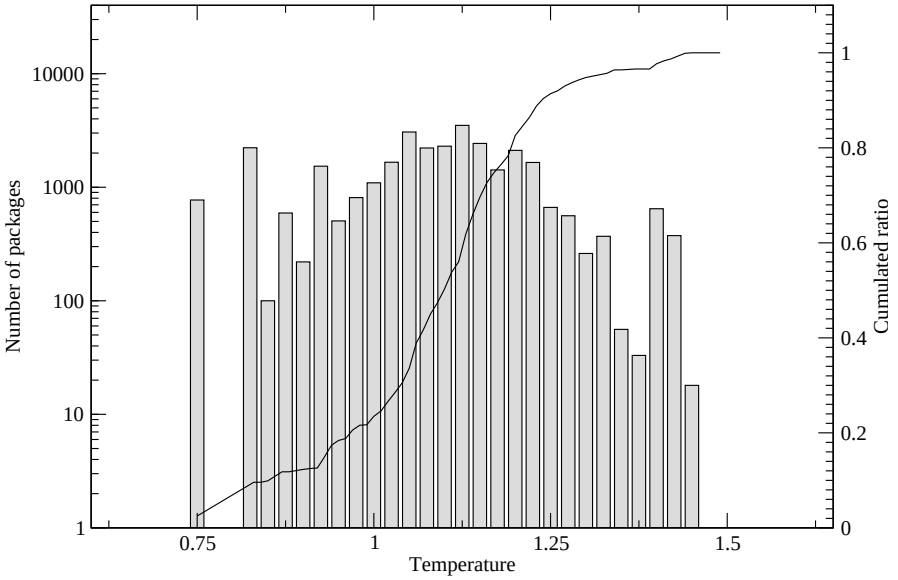


Fig. 6. Number of packages as a function of the "temperature" of the SAT problems corresponding to their installability problems.

properties of component collections, due to the large spectrum of languages, technologies, frameworks and interfaces spanned by a contemporary FOSS distribution.

Despite their algorithmic complexity, we have already performed large-scale tests indicating that the first of these properties can be mechanically checked in reasonable time. We continue similar investigations on the other properties.

We claim that providing efficient tools to check these properties is an essential step in order to ensure that the FOSS development model stays sustainable, and we suggest that researchers should look into the specificities brought by FOSS in the software engineering world.

References

1. Broy, M., Denert, E.: Software Pioneers: Contributions to Software Engineering. Springer-Verlag (2002)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison Wesley Professional (1997)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
4. Tuura, L.A.: Ignominy: tool for analysing software dependencies and for reducing complexity in large software systems. In: Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research. Volume 502. (2003) 684–686
5. Taylor, L., Tuura, L.: Ignominy: a tool for software dependency and metric analysis with examples from large HEP packages. In: Proceedings of CHEP'01. (2001)
6. Eklund, D.: The lib update/autoupdate suite. http://luau.sourceforge.net/ (2003–2005)
7. van der Storm, T.: Variability and component composition. In: Proceedings of the Eighth International Conference on Software Reuse (ICSR-8). (2004)
8. Silva, G.N.: Apt-howto. http://www.debian.org/doc/manuals/apt-howto/ (2004)
9. Mandriva: URPMI. http://www.urpmi.org/ (2005)
10. Niemeyer, G.: Smart package manager. http://labix.org/smart/ (2005)
11. Debian Group: Debian policy manual. http://www.debian.org/doc/debian-policy/ (1996–1998)
12. Bailey, E.C.: Maximum RPM, taking the Red Hat package manager to the limit. http://rikers.org/rpmbook/,http://www.rpm.org (1997)
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Volume 2919 of Lecture Notes in Computer Science., Springer (2004) 502–518
14. LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. Submitted to Journal of Theoretical Computer Science (2005)

Appendix E

Contribution to the VII Workshop de Software Livre (WSL06)

The following is the contribution of the WorkPackage2 Team to the VII Workshop de Software Livre (WSL06)

News from the EDOS project: improving the maintenance of free software distributions *

Jaap Boender [1] **, Roberto Di Cosmo** [1] **, Berke Durak** [2]
Xavier Leroy [2] **, Fabio Mancinelli** [1] **, Mario Morgado** [3]
David Pinheiro [3] **, Ralf Treinen** [4]**, Paulo Trezentos** [3] **, Jérôme Vouillon** [1]

[1] PPS, University of Paris 7, Firstname.Lastname@pps.jussieu.fr

[2]INRIA Rocquencourt, Firstname.Lastname@inria.fr

[3]Caixa Magica, Firstname.Lastname@caixamagica.pt

[4]LSV, ENS de Cachan, Firstname.Lastname@lsv.ens-cachan.fr

Abstract. *The EDOS research project aims at contributing to the quality assurance of free software distributions. This is a major technical and engineering challenge, due to the size and complexity of these distributions (tens of thousands of software packages). We present here some of the challenges that we have tackled so far, and some of the advanced tools that are already available to the community as an outcome of the first year of work.*
Keywords: *free software, open source software, dependency management, constraint satisfaction, rollback, EDOS project.*

1 Introduction

So-called *distribution editors* like Caixa Magica, Conectiva, Debian, Mandriva, RedHat, Suse, Ubuntu and many others try to offer some kind of reference viewpoint over the breathtaking variety of free and open source software (FOSS) available today: they take care of packaging, integrating and distributing tens of thousands of software packages, very few of them being developed in-house and almost all coming from independent developers. As a consequence, most FOSS distributions today simply rely on the general notion of a software *package*[1]: a bundle of files containing data, programs, and configuration information, with some metadata attached. Most of the metadata information is about *dependencies*, i.e., the relationships with other packages that may be needed in order to run or install a given package, or that conflict with its presence on the system.

How can one ensure the quality of a distribution? This problem, which is the focus of the European Sixth Framework Programme project EDOS (Environment for the development and Distribution of Open Source software), can essentially be divided into three main tasks:

Upstream tracking makes sure that a package in the distribution closely follows the evolution of the software development, almost always done by some team beyond direct control by the distributor.

*This work was supported by the EDOS Specific Targeted Research Project of the 6th European Union Framework Programme.
[1]Not to be mistaken for the software organizational units such as librairies, modules or classes.

Testing and integration makes sure that a program performs as expected in combination with other packages in the distribution. If this is not the case then bug reports should be sent to the upstream developer.

Dependency management makes sure that, in a distribution, packages can be installed and user installations can be upgraded when new versions of packages are produced, while respecting the constraints imposed by the dependency metadata.

Inside the EDOS project, work package 2 (WP2) is about dependency management. This task is surprisingly complex [Tuura 2003, Van der Storm 2004] due to the large number of packages present in a typical distribution and to the complexity and richness of their interdependencies. More specifically, our focus is on the issues related to dependency management for large sets of software packages, with a particular emphasis on maintaining consistency of a software distribution *on the repository side*, as opposed to maintaining a set of packages installed *on a client machine*. This choice is justified by the following observation: maintaining consistency of a distribution of software packages is *fundamental* to the quality and scalability of current and future distributions; yet, it is also an *invisible* task, since the smooth working it ensures on the end user side tends to be considered as normal and obvious as the smooth working of packet routing on the Internet. In other words, we are tackling an essential *infrastructure* problem that has long been ignored: while there are a wealth of client-side tools to maintain a user installation (`apt`, `urpmi`, `smart` and many others [Silva 2004, Mandriva 2005, Niemeyer 2005]), there are surprisingly little literature and publically available tools that address server-side requirements. We found very little significant prior work in this area, despite it being critical to the success of FOSS in the long term.

In this short paper we want to give an overview of some of the tools developed inside the EDOS Project that have the potential to improve the management of distributions from the point of view of dependencies. The paper is organized as follows. Section 2 contains a formal description of the main characteristics of a software package found in mainstream FOSS distributions as far as dependencies are concerned. The algorithmic aspects of solving the dependency constraints are underlined in Section 3. Section 4 describes the usage of some of these tools for a Linux distribution (Caixa Magica) and their extension of APT with a rollback feature. Some metadata exploration tools developed by the WP2 and taking into account the historical evolution of the repositories are described in Section 5.

2 Basic definitions

Every package management system [Debian 2005, Bailey 1997] takes to various extends into account the interrelationships among packages. We will call these relationships *requirements*. Several kinds of requirements can be considered. The most common one is a *dependency* requirement: in order to install package P_1, it is necessary that package P_2 be installed as well. Less often, we find *conflict* requirements: package P_1 cannot coexist with package P_2.

Some package management systems specialize these basic types of requirements by allowing to specify the *time frame* during which the requirement must be satisfied. For

example, it is customary to be able to express *pre-dependencies*, a kind of dependency stating that some package P_1 needs some package P_2 to be present on the system *before* P_1 can be installed [Debian 2005].

These notions can be made precise [Di Cosmo et al. 2006], and we refer the interested reader to that paper for a more detailed discussion. For the current presentation we do not need such a level of formal precision, and we will rely on the intuitive meaning of commonly used terms like package, dependency, conflict, repository and installation.

The first, most basic quality requirement for a distribution is that for each package P being part of a distribution there should exist at least one installation of the distribution that satisfies all dependency constraints and that contains P. Otherwise, P is useless: nobody will ever be able to install it without breaking the dependency constraints, which in turn breaks the package management system.

Definition 1 (Installability)**.** *A package π of a repository R is* installable *if there exists an installation I that contains π and that is* healthy*, i.e. with no broken dependencies.*

We say that a repository R is *trimmed* when every package of R is installable w.r.t. R. The intuition behind this terminology is that a non-trimmed repository contains packages that cannot be installed in any configuration. We call those packages *broken*, and they should be excluded from the repository.

Our first set of tools is able to formally check whether a repository is trimmed, and if not it explains which packages are not installable for which reason.

3 Algorithmic considerations

It is not obvious that checking a repository for broken packages is actually tractable in practice: due to the rich language allowed to describe package dependencies in the mainstream FOSS distributions, this task may involve verifications over a large number of packages. During our first investigations of these problems we have indeed already proven the following complexity result.

Theorem 1 (Package installability is an NP-complete problem)**.** *Checking whether a single package P can be installed, given a repository R, is NP-complete.*

Nevertheless, the actual instances of these problems, as found in real repositories, turn out to be quite simple in the average.

We implemented various checking tools, using custom solvers as well as a SAT solver [Eèn and Sörensson 2004] and CP solvers, and ran them over both the Debian pool (about 15,000 source packages giving 20,000 units, totaling 30,000 packages in the different distributions, including ”contrib” and ”non-free” packages) and the Mandriva Cooker distribution (around 5,000 packages). The execution time is completely satisfactory, and the tools found a number of non-installable packages in both distributions. These tools are available from the subversion server of the EDOS project at `http://www.edos-project.org/`.

Notice that, unlike scripts that are actually used in some distributions, these EDOS tools are *correct and complete*, that is, they find *all* broken packages, and *only* the broken

packages, and they are *highly efficient*, as they can analyze the whole Debian repository in just a couple of minutes.

You can of course simply go to the EDOS subversion repository and download the tools to run them on the command line, but we have also two real-world deployment examples which show how a distribution manager could use them in a production environment.

4 Deployment and usage of the tools at Caixa Magica

Caixa Magica is a Portuguese Linux distribution used nationwide in Portugal. It is used not only in schools and public administrations but also by companies and private citizens. It is based on the RPM format although it uses `apt` (namely `apt-rpm`) since 2004. As in other Linux distributions, it has FTP servers with the official software packages (RPMs) and servers with unofficial RPMs submitted by the community. The company encourages the submission of these unofficial packages since they are much more up-to-date than the stable and official ones. For that purpose Caixa Magica has created a website, named "ContribWare" (`http://contribware.caixamagica.pt/`), which maintains the submission of unofficial packages. ContribWare is now 3 months old and hundreds of packages have been submitted through it. One workflow-related problem of such systems is detecting the broken dependencies. The contributors who submit packages usually have a lot of software installed and thus may fail to identify dependencies. These dependencies can also be on not yet packaged software.

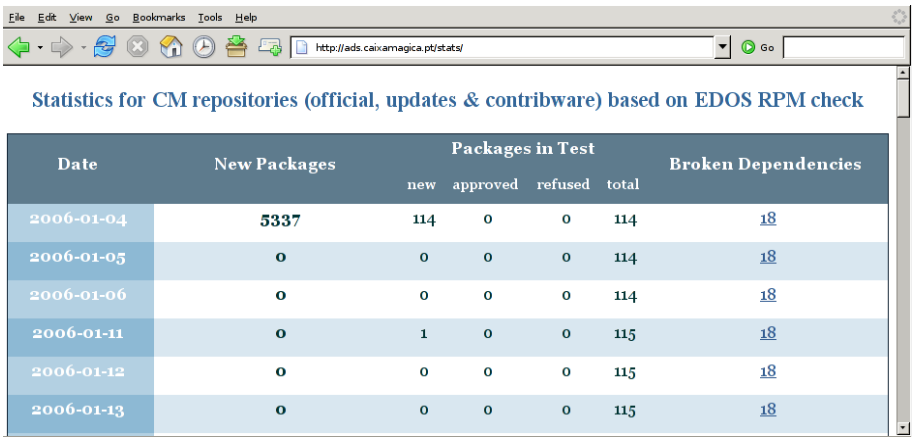
4.1 Description of graphical statistics interface

Using the WP2 `rpmcheck` tool, we were able to identify broken dependencies in ContribWare. A Python script has been developed for processing the information and displaying an HTML page containing a table summarizing the detected problems, as in Figure 1. The table has the following columns:

- New Packages: new packages added to the repository. It began with 5,337 packages.
- Packages in Test: packages that have been submitted to ContribWare by users and that have been moved to the "on test" state by Caixa Magica editors. This column has 4 sub-columns: *new*, *approved*, *refused* and *total*. Packages that are approved go to "New packages" and leave the "on test" state.
- Broken Dependencies: this column gives the number of packages with broken dependencies, with a link to a more detailed description. The latter gives the broken packages with their unsatisfied dependencies.

4.2 Apt rollback - extending package maintenance

The EDOS team is also developing some enhancements to the package management process. One of them is the APT rollback mechanism. As some upgrades are not always successful, customer requirements on quality assurance opened the need for implementing a rollback mechanism into `apt-rpm`. This mechanism relies on registering the requests for installation, upgrade, downgrade and removal of packages from the system as well as saving, in some situations, the packages' configuration files (depending on the operation



File Edit View Go Bookmarks Tools Help

http://ads.caixamagica.pt/stats/

Go

Statistics for CM repositories (official, updates & contribware) based on EDOS RPM check

Date	New Packages	Packages in Test				Broken Dependencies
		new	approved	refused	total	
2006-01-04	5337	114	0	0	114	18
2006-01-05	0	0	0	0	114	18
2006-01-06	0	0	0	0	114	18
2006-01-11	0	1	0	0	115	18
2006-01-12	0	0	0	0	115	18
2006-01-13	0	0	0	0	115	18

Figure 1. Daily log of Caixa Magica archives.

to be performed on the packages: upgrade, downgrade or removal). This mechanism permits to restore the system back to its state before any `apt` operation. For example, if an error is detected after an upgrade, the system can quickly restore its previous state. With every `apt-rpm` operation we save the following information:

- the package's name and version before the operation,
- the package's version after the operation (only for upgrades or downgrades),
- the operation's type (install, upgrade, downgrade or remove),
- a transaction ID,
- a timestamp and
- the package's configuration files.

If the operation is an upgrade, a downgrade or a removal, we query the package's metadata to check the existence of any configuration files, and if so we save them. Note that files that are saved are the ones available in the system, not the package's original configuration files, so that when a rollback is performed we ensure that we restore the configuration files as they were at the time of the operation, including user modifications.

A rollback is basically the inverse of a transaction. It includes the inverse package operation (downgrade for an upgrade, removal for an installation, etc.) as well as the restoration of the package's configuration files (if necessary) as shown in Table 1:

Operation	Rollback operation	Action taken with the Configuration Files
`install`	`remove`	None
`remove`	`install`	Restore configuration
`upgrade`	`downgrade`	Restore configuration files
`downgrade`	`upgrade`	Restore configuration files

Table 1. Rollback operation relationships.

We implemented this functionality into `libapt`, thus ensuring that tools like `synaptic` also register every operation performed and we added two more commands to `apt-get`:

- `apt-get rollback-hist`: For displaying the history of operations with their transaction IDs

- `apt-get rollback <transaction id>`: For rolling back the operations performed in the given transaction.

5 EDOS Tools for Exploring the Debian History

The `history` tool allows command-line exploration of the Debian metadata using an algebraic query language. Daily metadata is extracted from the archives on `snapshot.debian.net` and stored in a MySQL database. However, despite the data being stored in a structured manner and being appropriately indexed, performance of even simple SQL queries (such as finding the set of immediate potential dependencies of a set of packages) is poor. This is partly due to the complex nature of metadata, which is, due to disjunctive dependencies, not a graph but a a hypergraph, whose relational representation requires multiple levels of indirection. Also, it is well-known that standard SQL cannot handle transitive closures. Hence we have opted for using the SQL database only as an off-line storage engine; `history` loads the whole database into RAM, and can then answer complex queries efficiently. We now give a very short introduction to the query language.

The tool works as a classic read-evaluate-print loop. Expressions or directives are entered and their results printed. The basic data types handled by `history` are units, packages, sources, sets of the above, dates, integers and booleans. Care has been taken to provide concise notation for describing these. The basic features of a functional language with strict evaluation are provided. The usual Boolean operations on sets (intersection $\&$, union $|$ and difference $\backslash$) are allowed. Sets can be filtered by regular expressions or user-defined functions. With operators such as `exists` and `for_all`, this effectively provides first-order quantified queries. Besides directives that print the metadata in human-readable form, various operators are provided so that the metadata can be used in complex expressions. The metadata is historically represented as functions which map dates to sets of packages. For instance, if p is a package, `provides`(p) is the set of units provided by p, and if s is a set of packages, `closure`(s) is the dependency closure of s : all packages that packages of s might need to run are contained in s – but due to disjunctive dependencies, s will usually contain extraneous packages, and due to conflicts, it might not be possible to install all packages of s. Thus, the dependency solver of `debcheck` and `rpmcheck` has been integrated into `history` as an operator `install`(p_1, p_2) which computes a set p of co-installable packages such that $p_1 \subseteq p \subseteq p_2$. Table 2 gives an overview of the available operators.

We are developing a web version of `history` with an interface similar to that of `ara`[2]. An early prototype integrating results from the dependency solver and the historical metadata database, called `anla`, is available at `http://brion.inria.fr/anla/`, see Figure 2 for a screenshot.

6 Conclusions

We hope that the efficient and formally based tools developed by the EDOS project will be soon adopted by distribution editors to improve their production cycle.

[2] `ara` is a text-based search engine for Debian packages which can compute Boolean combinations of field-restricted regular expressions, available at `http://ara.edos-project.org/`

Operator	Meaning
$s\&t, s\|t, s \setminus t$	Boolean set intersection, union and difference
`provides`(p)	Set of units provided by a package
`conflicts`(p)	Set of packages that conflict with a package
`closure`(p)	Dependency closure of a package
`source`(p)	Source of a package
`unit`(p)	Unit of a package
`latest`(u)	Latest version of a unit
`versions`(u)	All the versions of a unit
`what_provides`(u)	The set of packages that provide a unit
`replaces`(p)	The set of packages replaced by a package
`install`(p, q)	Returns an installation of the set of packages p inside the set q
`member`(x, s)	True when the element `x` is a member of the set `s`
`filter`(s, f)	Return the elements of s for which $f(s)$ is true.
`exists`(s, f)	True when $f(s)$ is true for one element of s
`for_all`(s, f)	True when $f(s)$ is true for all elements of s

Table 2. Operators. Most operators are overloaded to work on sets. Functions are first-class values with lexical scoping that can be defined anonymously.

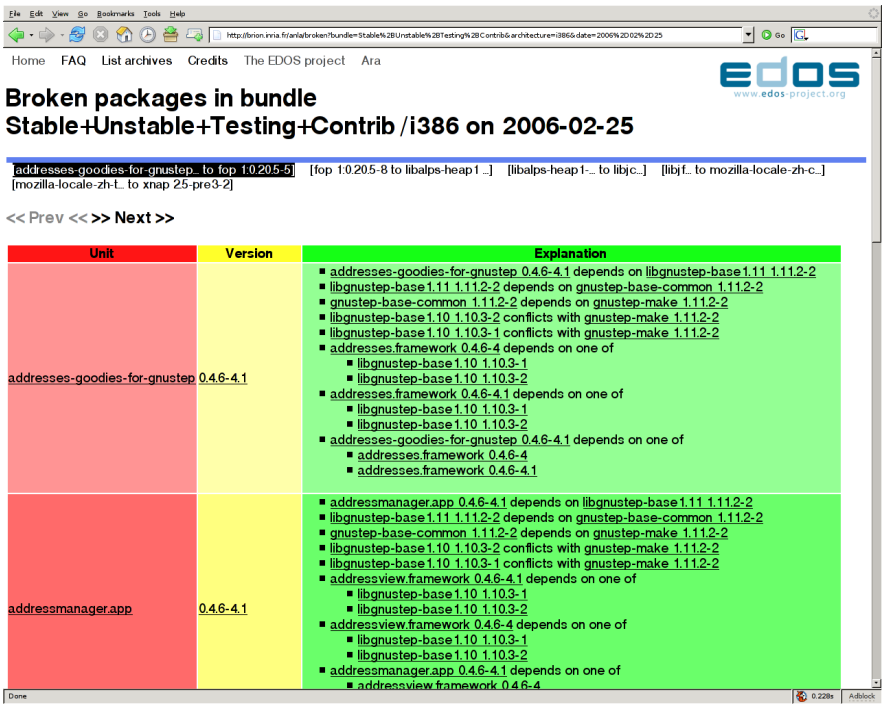


Figure 2. Checking status of Debian archives. Most broken packages owe their status to dependency on packages that are not in their archives, for instance a package in unstable that depends on a package in stable is broken. To filter out these uninteresting cases, we have merged archives into a bundle before launching the checker. All the displayed metadata information is fully hyperlinked.

References

Edward C. Bailey. Maximum RPM, taking the Red Hat package manager to the limit. http://rikers.org/rpmbook/,http://www.rpm.org, 1997.

Manfred Broy and Ernst Denert. *Software Pioneers: Contributions to Software Engineering*. Springer-Verlag, 2002.

Debian Group. Debian policy manual. http://www.debian.org/doc/debian-policy/, 1996–2005.

David Eklund. The lib update/autoupdate suite. http://luau.sourceforge.net/, 2003–2005.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli and Jérôme Vouillon. *Maintaining large software distributions: new challenges from the FOSS era*. FRCSS06, Vienna, 1st April 2006.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *Submitted to Journal of Theoretical Computer Science*, 2005.

Mandriva. URPMI. http://www.urpmi.org/, 2005.

Gustavo Niemeyer. Smart package manager. http://labix.org/smart/, 2005.

Gustavo Noronha Silva. Apt-howto. http://www.debian.org/doc/manuals/apt-howto/, 2004.

Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

L. Taylor and L. Tuura. Ignominy: a tool for software dependency and metric analysis with examples from large HEP packages. In *Proceedings of CHEP'01*, 2001.

L. A. Tuura. Ignominy: tool for analysing software dependencies and for reducing complexity in large software systems. In *Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, volume 502, pages 684–686, 2003.

Tijs van der Storm. Variability and component composition. In *Proceedings of the Eighth International Conference on Software Reuse (ICSR-8)*, 2004.