

# News from the EDOS project: improving the maintenance of free software distributions \*

Jaap Boender<sup>1</sup>, Roberto Di Cosmo<sup>1</sup>, Berke Durak<sup>2</sup>, Xavier Leroy<sup>2</sup>  
Fabio Mancinelli<sup>1</sup>, David Pinheiro<sup>3</sup>, Paulo Trezentos<sup>3</sup>, Mario Morgado<sup>3</sup>, Jérôme Vouillon<sup>1</sup>

<sup>1</sup> PPS, University of Paris 7, `Firstname.Lastname@pps.jussieu.fr`

<sup>2</sup> INRIA Rocquencourt, `Firstname.Lastname@inria.fr`

<sup>3</sup> Caixa Magica, `Firstname.Lastname@caixamagica.pt`

**Abstract.** *The EDOS project is a research effort whose goal is to contribute to ensure the quality of a free software distribution. This is a major technical and engineering challenge, owing to the size and complexity of these distributions (tens of thousands of software packages).*

*We present here some of the challenges that we have already tackled, 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, EDOS project*

## 1. Introduction

The so-called *distribution editors*, Conectiva, Debian, Mandriva, RedHat, Suse, Caixa Magica, Ubuntu and so 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 being developed in-house and almost all coming from independent developers. As a consequence, most FOSS distribution today simply rely on the general notion of software *package*<sup>1</sup>: 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.

In figure 1 we give an overview of a typical FOSS process: 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` constitute projects, even if

---

\*This work was supported by the EDOS Specific Targeted Research Project of the 6th European Union Framework Programme.

<sup>1</sup>Not to be mistaken for the software organizational unit present in many modern programming languages.

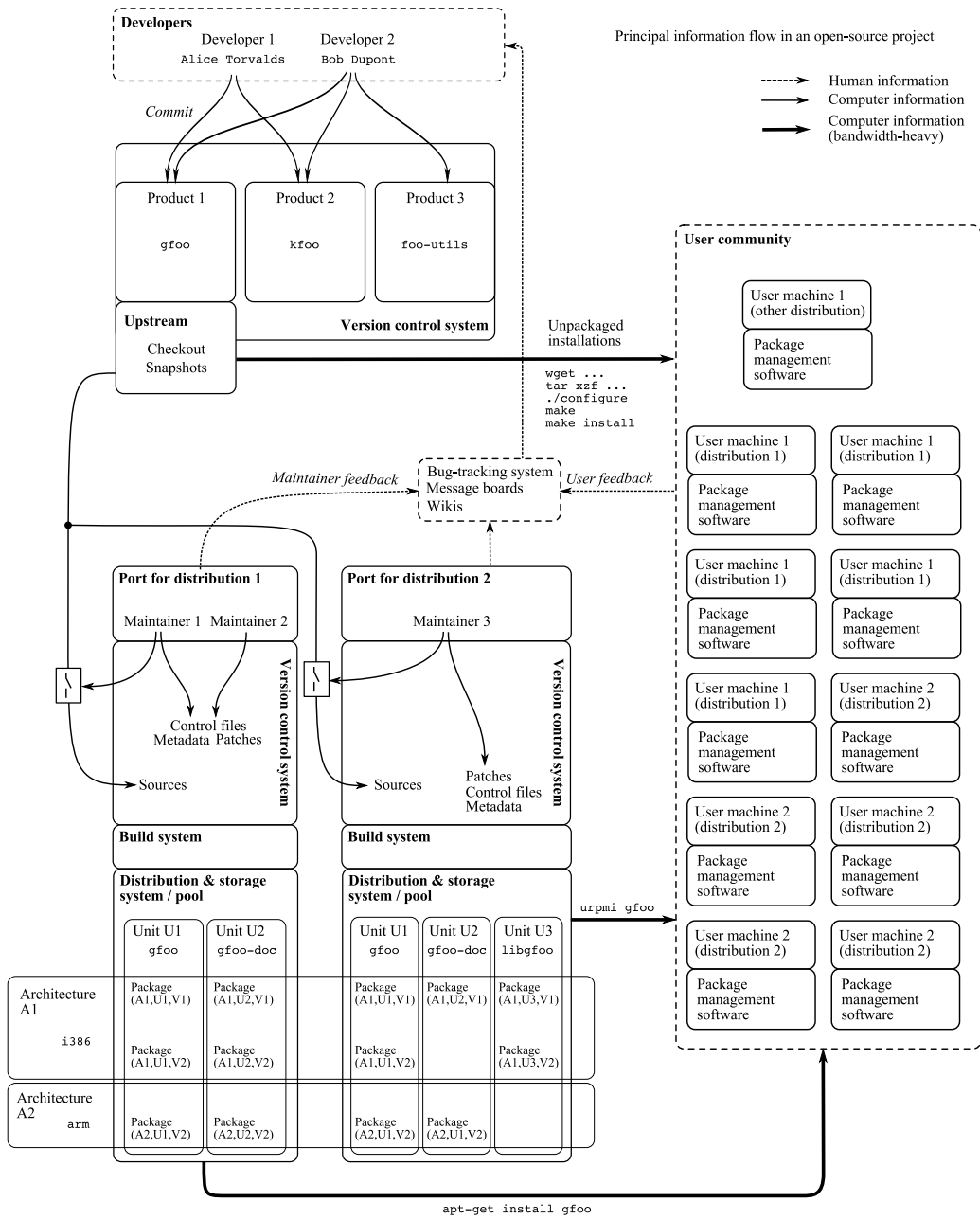


Figure 1. Major flow of information in a FOSS project.

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 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 [Tuu03, TT01] or getting source code updates from developers [Ekl05] 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`, `smart` 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.

How can one ensure the quality of a distribution? This problem, which is the focus of the European FP6 project EDOS (Environment for the development and Distribution of Open Source software), can be essentially 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.

Inside the EDOS project, the WP2 team is specifically targeting the third one: dependency management. This task is surprisingly complex [Tuu03, vdS04], 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 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 [Sil04, Man05, Nie05]), 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.

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 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. An overview of the tools is given in section ??, followed by conclusions in section 6..

## 2. Basic definitions

Every package management system [DG98, Bai97] 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 [DG98].

These notions can be made formally precise, as we did in [?], and we refer the interested reader to that paper for a more detailed discussion, but for the sake of the current presentation, we do not need that 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 satisfy 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  $\pi$  of a repository  $R$  is installable if there exists an installation  $I$  that contains  $\pi$  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 behave as if they were not part of the repository.

Our first set of tools is able to formally check whether a repository is trimmed, and if not, it details the errors, and explains them.

### 3. Algorithmic considerations

It is really not evident 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 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.*

Nevertheless, in practice 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 [ES04] and CP solvers, and ran them 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 tools found a number of non-installable packages in both distributions. These tools are available from the subversion server of the EDOS project <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 few 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 that 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 nation wide. It is used not only in schools and public administration but also by private citizens and companies. It is based in RPMs although it use apt-get (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 that were submitted by the community.

We encourage the submission of this unofficial packages since they are much more updated then the stable and official ones. For that purpose we created a website that maintains the submission of unofficial packages and was named “Contribware” (<http://contribware.caixamagica.pt>). ContribWare is now 3 months and hundreds of packages had been submitted through it.

One problem of such system to control the RPM workflow was detecting the broken dependencies. The power users that submit packages have most of the times a lot of software installed and do not identify potential requirements that are not match by RPMs in the repository.

##### **4.1. Description of graphical statistics interface**

Using WP2 rpmcheck tool, we were able to identify in the command line what dependencies were broken in the unofficial repository. Then, was developed a Python script that processes the information and draw a table with the problems detected . This tool generate an HTML page with a table like the one depict in figure 2.

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 had been submitted to ContribWare by the Caixa Magica users and had been moved to the state “on test” by Caixa Magica editors. This column has 4 sub-columns: *new*, *approved*, *refused* and *total*. Packages that are approved goes to “New packages” and leave the “on test” state.
- **Broken Dependencies:** the number of packages with broken dependencies is presented and a link to a more detailed description is presented. In the detailed description we can check which package is broken and the missing dependency.

##### **4.2. Apt rollback - extending package maintenance**

The EDOS team is also developing enhancements in different fields. One of them is apt rollback mechanism.

As some upgrades are not always successful and customer requirements on quality assurance opened the need for implementing this rollback mechanism into apt-rpm. This mechanism relies on registering the instalation, upgrade, downgrade and removal of

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
2006-01-14	0	1	0	1	115	18
2006-01-16	0	0	0	0	114	18
2006-01-17	0	0	0	0	114	18
2006-01-18	10	0	0	0	114	19
2006-01-19	0	0	0	0	114	19
2006-01-20	0	0	0	0	114	19
2006-01-21	0	0	0	0	114	19
2006-01-22	0	0	0	0	114	19
2006-01-23	0	0	0	0	114	19
2006-01-24	0	0	0	0	114	19
2006-01-25	0	0	0	0	114	19
2006-01-26	0	0	0	0	114	19
2006-01-27	1	0	0	0	114	19
2006-01-29	0	0	0	0	114	19
2006-01-31	0	11	0	0	125	19
2006-02-01	0	3	0	1	127	19
2006-02-02	0	5	0	0	132	19

Figure 2. Daily log of Caixa Magica archives.

any package into the system as well as saving, in certain situations, the package's configuration files (depending if the operation of the package implies an upgrade, a downgrade or a removal).

The main goal is to be able restore the system back to the state it was before the apt operation so that when an error is detected in the system after an upgrade for example, we can quickly restore it back to it's original state. In every operation performed by the apt-rpm we save the following information:

- package name
- package version
- package new version (only relevant in upgrades or downgrades)
- operation type (install / upgrade / downgrade /remove)
- transaction id
- time and date
- package's configuration files

If the operation is an upgrade, downgrade or a removal, we require the 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 install the configuration files that were being used by the packages at the time of the operation.

A rollback is basically the opposite operation of what was registered in a certain transaction id 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 `synaptic` also registers every operation performed and added two more options to the `apt-get` command line:

- `apt-get rollback-hist` - For displaying the history of operations as well as the transaction id's
- `apt-get rollback <transaction id>` : for rolling back the operation's performed in transaction id

## 5. The EDOS Debian History tool

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,  $(aspell - t1, 0.02 - 5)$  is a package. An *archive* is then a function that maps time to sets of packages<sup>2</sup>

The history tool allows command-line exploration of the Debian metadata using an algebraic syntax. Daily metadata is extracted that from the archives on `snapshot.debian.net`) and stored in a MySQL database. However, despite the data being stored in structured manner and appropriately indexed, performance of even simple SQL queries (such as finding the set of first-order dependencies of a package) is poor. This is due to the peculiar nature of Debian metadata, which is not even a graph but, due to disjunctive dependencies, a hypergraph. Also, it is well-known that standard

<sup>2</sup>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.

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 do complex queries in a fast manner. 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 in providing concise notation for describing these. 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. Variable names start with a '\$' and can be assigned with an expression like `$name <- expression`. Local binding is possible with constructions of the form `let$name = expression1inexpression2`. More complex data types include lists, arrays and functions – functions are written as `x -> expr`. Archive contents are accessed by giving the archive ID (an integer that can be obtained with the `#list_archives` directive). This returns a function that maps dates to sets of packages. For instance, `$archives 3 2005-11-02` returns the set of packages contained in `debian/stable/main/binary-i386` on November the 2nd, 2005. The usual boolean operations on sets (intersection `&`, union `|` and difference `\`) are allowed. A set can be filtered by a regular expression (that acts on the unit name) with an expression such as `x /regexp/`. Operators are provided for accessing the metadata. For instance, if `p` is a package, `provides(p)` is the set of units provided by `p`. Operators are overloaded where possible, hence if `P` is a set of packages, `provides(P)` is union for `p` ranging over `P` of `provides(p)`. Variables can be assigned. Quantification is also possible with the help of predicates like `for_all` and `exists`.

As an example, the following code fragment defines a function `$lost` that takes two dates  $t$  and  $t'$  and computes the “lost functionality” between them. If  $A_t$  (resp.  $A_{t'}$ ) is the set of packages of the archive on date  $t$  (resp.  $t'$ ), lost functionality is defined as the set of packages  $p \in A_t$  such that the following conditions hold:

- $p$  is not in  $A_{t'}$ ,
- No package  $p' \in A_{t'}$  replaces  $p$ ,
- No package  $p' \in A_{t'}$  has the same source as  $p$ .

```
\$lost <-
  \$start ->
  \$stop ->
  let \$architecture = "binary-i386" in
  let \$a = \$find_archive([ ["debian"];
                             ["stable";"unstable";"testing"];
                             ["main";"contrib";"non-free"];
                             [\$architecture] ]) in
  let \$x = \$a \$start in
  let \$y = \$a \$stop in
  let \$units2 = unit(\$y) in
```

Operator	Meaning
<code>provides(p)</code>	Set of units provided by a package
<code>conflicts(p)</code>	Set of packages that conflict with a package
<code>closure(p)</code>	Dependency closure of a package
<code>source(p)</code>	Source of a package
<code>unit(p)</code>	Unit of a package
<code>latest(u)</code>	Latest version of a unit
<code>versions(u)</code>	All the versions of a unit
<code>what_provides(u)</code>	The set of packages that provide a unit
<code>replaces(p)</code>	The set of packages replaced by a package
<code>install(p1.p2)</code>	Returns an installation of the set of packages p1 inside the set p2
<code>member(x,s)</code>	True when the element x is a member of the set s

Table 2. Operators.

```

let \sources2 = unit(source(\y)) in
let \replaces2 = replaces(\y)
in
filter( (\x \ y),
  \p1 ->
    (not member(\p1, \replaces2))
    and (not member(unit(\p1), \units2))
    and (not member(unit(source(\p1)), \sources2)) );

```

Finally, the dependency solver has been integrated into history. An operator `install( $p_1, p_2$ )` computes a set  $p$  of co-installable packages such that  $p_1 \subseteq p \subseteq p_2$ .

One might argue that augmenting the toplevel of a language with a few definitions and functions for handling Debian packages would be enough and that this tool is unnecessary. However this requires that the user be fluent in that programming language, or at least its type system. Furthermore, the simplified syntax and the overloading makes querying the system simple, while strongly typed native-code compilation makes it fast. Also, we are planning a web version of history with an interface similar to `ara`, which is a text-based search engine for Debian packages that can compute boolean combinations of field-restricted regular expressions. While command-line and GTK versions of `ara` are available in Debian stable (packages `ara` and `xara-gtk`), a web interface is hosted at <http://ara.edos-project.org/>. In fact, `ara` is an ancestor of `history`. An early prototype of a web interface integrating results from the dependency solver and the historical metadata database, called `anla`, is available at <http://brion.inria.fr/anla/>, see figure 3 for a screenshot.

## 6. Conclusions

We hope that the tools developed by the EDOS project will be soon largely adopted by most distribution editors, to improve their production cycle, as the high efficiency of these tools, together with their formal basis, make them immediately deployable.



Figure 3. Checking status of Debian archives.

## 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/>, 1996–1998.
- 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.