Formal Aspects of Free and Open Source Software Components*

A Short Survey

Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS
roberto@dicosmo.org, {treinen,zack}@pps.univ-paris-diderot.fr

Abstract. Free and Open Source Software (FOSS) distributions are popular solutions to deploy and maintain software on server, desktop, and mobile computing equipment. The typical deployment method in the FOSS setting relies on software *distributions* as vendors, *packages* as independently deployable components, and *package managers* as upgrade tools. We review research results from the past decade that apply formal methods to the study of inter-component relationships in the FOSS context. We discuss how those results are being used to attack both issues faced by users, such as dealing with upgrade failures on target machines, and issues important to distributions such as quality assurance processes for repositories containing tens of thousands, rapidly evolving software packages.

1 Introduction

Free and Open Source Software [47], or FOSS, is used daily, world-wide, to manage computing infrastructures ranging from the very small, with embedded devices like Android-based smart phones, to the very big, with Web servers where FOSS-based solutions dominate the market. From the outset, most FOSS-based solutions are installed, deployed, and maintained relying on so-called *distributions*. The aspect of software distributions that will interest us in this paper is that they provide a *repository*: a typically large set of software *packages* maintained as software components that are designed to work well together. Software distributions, like for instance GNU/Linux distributions, have in fact additional aspects that are crucial but which are not considered in this work, like for instance an installer that allows a user to install an initial system on a blank machine, or infrastructure for interaction between users and developers like a bug tracking system.

While specific technologies vary from distribution to distribution, many aspects, problems, and solutions are common across distributions. For instance, packages have expectations on the deployment context: they may require other

* Work partially supported by Aeolus project, ANR-2010-SEGI-013-01, and performed at IRILL, center for Free Software Research and Innovation in Paris, France, www.irill.org

E.Giachino et al. (Eds.): FMCO 2012, LNCS 7866, pp. 216–239, 2013.

packages to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other packages—declaring this fact by means of *conflicts*. Dependencies and conflicts are captured as part of *package metadata*. Here as an example showing the popular Firefox web browser as a package in the Debian distribution:

```
Package: firefox
Version: 18.0.1-1
Depends: libc6 (>= 2.4), libgtk2.0-0 (>= 2.10), libstdc++6,
  fontconfig, procps, xulrunner-18.0, libsqlite3-0, ...
Suggests: fonts-stix | otf-stix, mozplugger,
  libgssapi-krb5-2 | libkrb53
Conflicts: mozilla-firefox (<< 1.5-1)
Provides: www-browser, gnome-www-browser
```

A couple of observations are in order. First, note how the general form of inter-package relationships (conflicts, dependencies, etc.) is that of propositional logic formulae, having as atoms predicates on package names and their versions. Second, we have various degrees of dependencies, strong ones (like "Depends") that must be satisfied as deployment preconditions and weak ones (like "Suggests" and "Recommends", the latter not shown). Finally, we also observe an indirection layer in the package namespace implemented by "Provides". Provided packages are sometimes referred to as *features*, or *virtual packages* and mean that the providing package can be used to satisfy dependencies for—or induce conflicts with—the provided name.

To maintain package assemblies, semi-automatic *package manager* applications are used to perform package installation, removal, and upgrades on target machines—the term *upgrade* is often used to refer to any combination of those actions. Package managers incorporate numerous functionalities: trusted retrieval of components from remote repositories, planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*), user interaction to allow for interactive tuning of upgrade plans, and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are enco

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab [...]
The following actions will resolve these dependencies:
Remove the following packages: gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages: libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)] [...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Fig. 1. Attempt to install a disk space monitoring utility (called *baobab*) using the Aptitude package manager. In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in [4] a trivial alternative solution exists that minimizes system changes: remove a couple of dummy "meta" packages.

managers. A related issue, that we will also discuss in the following, is that of providing expressive languages that allow users of package managers to express their preferences, e.g. the demand to minimize the size occupied by packages installed on their machines.

Distribution editors, on the other hand, face the challenging task of avoiding inconsistencies in huge package archives. A paradigmatic example of inconsistency that they should avoid is that of shipping *uninstallable packages*, i.e. packages that, no matter what, cannot be installed on user machines because there is no way to satisfy their dependencies and conflicts. Consider for instance the following (real) example involving the popular Cyrus mail system:

Package: `cyrus-common-2.2`	**Package**: `cyrus-common-2.4`
Version: `2.4.12-1`	**Version**: `2.4.12-1`
Depends: `cyrus-common-2.4`	**Conflicts**: `cyrus-common-2.2`

It is easy to verify that it is not possible to install the above `cyrus-common-2.2` package—a dummy package made to ease upgrades to Cyrus 2.4—out of any package repository that also contains the `cyrus-common-2.4` package shown in the example. Even worse, it can be shown that the issue is not transitional, i.e. the team responsible for `cyrus-common-2.2` (its *maintainers*) cannot simply wait for the issue to go away, they have to manually fix the metadata of their package so that the cause of the uninstallability goes away. The challenge here is that, while it is easy to reason on simple cases like this one, distribution editors actually need semi- or fully-automated tools able to spot this kind of quality assurance issues and point them to the most likely causes of troubles.

Paper Structure. In the following we provide a short summary of research from the past decade on the formal aspects of FOSS packages. We first present, in Sect. 2, different formal models able to capture the parts of package metadata

that are relevant to attack both issues faced by users and by distributions. Then, in Sect. 3, we give an overview of results that foster the development of complete and expressive package managers that would provide a better package management experience to users. Finally, in Sect. 4, we do the same with research results that have been used to develop and deploy semi-automated quality assurance tools used daily by editors of popular FOSS distributions to assess the quality of their package repositories.

2 Formal Package Models

Different formal treatments of packages and their relationships are needed for different purposes. Two main approaches have been devised: a syntactic (or *concrete*) one which captures the syntax of inter-package relationships, so that they can be treated symbolically, similarly to how package maintainers reason about them. We will use such an approach to reason about the future evolution of repositories (see Sect. 4), taking into account yet unknown package versions.

A more abstract package model is useful too, in order to make the modeling more independent from specific component technologies and their requirement languages. We will use this kind of modeling to recast the problem of verifying package installability as a SAT problem (see Sect. 2.3).

2.1 Concrete Package Model

A concrete package model, originally inspired by Debian packages, has been given in [5] and further detailed in [6]. In this model packages are captured as follows:

Definition 1 (Package). *A* package (n, v, D, C) *consists of*

- *a* package name $n \in \mathrm{N}$,
- *a* version $v \in \mathrm{V}$,
- *a set of dependencies* $D \subseteq \wp(\mathrm{N} \times \textsc{Con})$,
- *a set of conflicts* $C \subseteq \mathrm{N} \times \textsc{Con}$,

where N is a given set of possible package names, V a set of package versions, and Con a set of syntactic constraints on them like $\top$, $= v$, $> v$, $\leq v$, ... The intuition is that dependencies should be read as *conjunctions of disjunctions*. For example: $\{\{(p, \geq 1), (q, = 2)\}, \{(r, < 5)\}\}$ should be read as $((p \geq 1) \vee (q = 2)) \wedge (r < 5)$. Starting from this intuition, the expected semantics of package constraints can be easily formalized.

Notation 1. *Given a package p we write p.n (resp. p.v, p.D, p.C) for its name (resp. version, dependencies, conflicts).*

Repositories can then be defined as package sets, with the additional constraint that name/version pairs are unambiguous package identifiers:

Table 1. Sample package repository

```
Package: a                Package:   b            Package: d
Version: 1                Version:   2            Version: 3
Depends: b (≥ 2) | d      Conflicts: d

Package: a                Package:   c            Package: d
Version: 2                Version:   3            Version: 5
Depends: c (> 1)          Depends:   d (> 3)
                          Conflicts: d (= 5)
```

Definition 2 (Repository). *A* repository *is a set of packages, such that no two different packages carry the same name and version.*

A pair of a name and a constraint has a meaning with respect to a given repository R, the precise definition of which would depend on the formal definition of constraints and their semantics:

Notation 2. *Given a repository R, $n \in \mathrm{N}$ and $c \in \textsc{Con}$, we write $[[(n, c)]]_R$ for the set of packages in R with name n and whose version satisfies the constraint c.*

We can then finally capture the important notions of installation and of (co-)installability:[1]

Definition 3 (Installation). *Let R be a repository. An R-installation is a set of packages $I \subseteq R$ such that $\forall p, q \in I$:*

abundance *for each element $d \in p.D$ there exists $(n, c) \in d$ and a package $q \in I$ such that $q \in [[(n, c)]]_R$.*
peace *for each $(n, c) \in p.C$: $I \cap [[(n, c)]]_R = \emptyset$*
flatness *if $p \neq q$ then $p.n \neq q.n$*

Definition 4 (Installability). *$p \in R$ is R-installable if there exists an R-installation I with $p \in I$.*

Definition 5 (Co-Installability). *$S \subseteq R$ is R-co-installable if there exists an R-installation I with $S \subseteq I$.*

Example 1 (Package Installations). Consider the repository R shown in Table 1. The following sets are not R-installations:

– R as a whole, since it is not flat;

[1] We remind that this is a *specific* concrete package model, inspired by Debian packages. Therefore not all installation requirements listed here have equivalents in *all* component technologies. Most notably the presence of the flatness property varies significantly from technology to technology. As discussed in [4,6] this does not affect subsequent results.

– $\{(a,1),(c,3)\}$, since both a's and b's dependencies are not satisfied;
– $\{(a,2),(c,3),(d,5)\}$, since there is a conflict between c and d.

The following sets are valid R-installations: $\{(a,1),(b,2)\}$, $\{(a,1),(d,5)\}$. We can therefore observe that the package $(a,1)$ is R-installable, because it is contained in an R-installation.

The package $(a,2)$ is not R-installable because any installation of it must also contain $(c,3)$ and consequently $(d,5)$, which will necessarily break peace. □

2.2 Abstract Package Model

A more abstract package model [43] was the basis for several of the studies discussed in the present work. The key idea is to model repositories as non mutable entities, under a closed world assumption stating that we know the set of all existing packages, that is that we are working with respect to a given repository R.

Definition 6. *An* abstract repository *consists of*

– *a set of packages* P,
– *an anti-reflexive and symmetric* conflict relation $C \subseteq P \times P$,
– *a* dependency function $D\colon P \longrightarrow \wp(\wp(P))$.

The nice properties of peace, abundance, and (co-)installability can be easily recast in such a model.

The concrete and abstract models can be related. In particular, we can translate instances of the concrete model (easily built from real-life package repositories) into instances of the more abstract model, preserving the installability properties. To do that, the main intuition is that (concrete) package constraints can be "expanded" to disjunctions of all (abstract) packages that satisfy them. For example, if we have a package p in versions 1, 2, and 3, then a dependency on $p \geq 2$ will become $\{\{(p,2),(p,3)\}\}$. For conflicts, we will add a conflict in the abstract model when either one of the two (concrete) packages declare a conflict on the other, or when we have two packages of the same name and different versions. The latter case implements the flatness condition. Formally:

Notation 3. *Let R be a repository in the concrete model. We can extend the semantics of pairs of names and constraints to sets as follows:*

$$[[\{(n_1,c_1),\ldots,(n_m,c_m)\}]]_R = [[(n_1,c_1)]]_R \cup \ldots \cup [[(n_m,c_m)]]_R$$

Definition 7 (Concrete to Abstract Model Translation). *Let R be a repository in the concrete model. We define an abstract model $R_a = (P_a, D_a, C_a)$.*

– *P_a: the same packages as in R*
– *We define the dependency in the abstract model:*

$$D_a(p) = \{[[\phi]]_R \mid \phi \in p.D\}$$

– *We define conflicts in the abstract model:*

$$\begin{aligned} C_a = {} & \{(p_1,p_2) \mid p_1 \in [[p_2.C]]_R \vee p_2 \in [[p_1.C]]_R\} \\ & \cup \{(p_1,p_2) \mid p_1.n = p_2.n \wedge p_1.v \neq p_2.v\} \end{aligned}$$

Install `libc6` version `2.3.2.ds1-22` in

Package: libc6
Version: 2.2.5-11.8

Package: libc6
Version: 2.3.5-3

Package: libc6
Version: 2.3.2.ds1-22
Depends: libdb1-compat

Package: libdb1-compat
Version: 2.1.3-8
Depends: libc6 (>= 2.3.5-1)

Package: libdb1-compat
Version: 2.1.3-7
Depends: libc6 (>= 2.2.5-13)

$\Rightarrow$

$$
\begin{aligned}
&\texttt{libc6}_{2.3.2.ds1-22} \\
&\wedge \\
&\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
&\wedge \\
&\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.3.5-3}) \\
&\wedge \\
&\neg(\texttt{libc6}_{2.3.5-3} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
&\wedge \\
&\neg(\texttt{libdb1-compat}_{2.1.3-7} \wedge \texttt{libdb1-compat}_{2.1.3-8}) \\
&\wedge \\
&\texttt{libc6}_{2.3.2.ds1-22} \rightarrow \\
&(\texttt{libdb1-compat}_{2.1.3-7} \vee \texttt{libdb1-compat}_{2.1.3-8}) \\
&\wedge \\
&\texttt{libdb1-compat}_{2.1.3-7} \rightarrow \\
&(\texttt{libc6}_{2.3.2.ds1-22} \vee \texttt{libc6}_{2.3.5-3}) \\
&\wedge \\
&\texttt{libdb1-compat}_{2.1.3-8} \rightarrow \texttt{libc6}_{2.3.5-3}
\end{aligned}
$$

Fig. 2. Example: package installability as SAT instance

2.3 On the Complexity of Installability

Now that we have rigorously established the notion of package (co-)installability, it is legitimate to wonder about the complexity of deciding these properties. Is it "easy enough" to automatically identify non-installable packages in large repositories of hundreds of thousands of packages? The main complexity result, originally established in [43], is not encouraging:

Theorem 1. *(Co-)installability is NP-hard (in the abstract model).*

The gist of the proof is a bidirectional mapping between boolean satisfiability (SAT) [19] and package installability. For the forward mapping, from packages to SAT, we use one boolean variable per package (the variable will be true if and only if the corresponding package is installed), we expand dependencies as implications $p \rightarrow (r_1 \vee \cdots \vee r_n)$ where r_i are all the packages satisfying the version constraints, and encode conflicts as $\neg(p \wedge q)$ clauses for every conflicting pair (p, q). Thanks to this mapping, we can use SAT solvers for checking the installability of packages (see Fig. 2 and Sect. 4).

The backward mapping, from SAT to package installability, can be established considering 3-SAT instances, as detailed in [30].

Given that the proof is given for the abstract model, one might wonder to which kind of concrete models it applies. The question is particularly relevant to know whether dependency solving in the context of specific component technologies can result in corner cases of unmanageable complexity or not. Several instances of this question have been answered in [4], considering the common features of several component models such as Debian and RPM packages, OSGi [45] bundles, and Eclipse plugins [20,15]. Here are some general results:

- Installability is NP-complete provided the component model features conflicts and disjunctive dependencies.
- Installability is in PTIME if the component model does *not* allow for conflicts (neither explicitly, nor implicitly with clauses like Eclipse's "singleton").
- Installability is in PTIME if the component model does not allow for disjunctive dependencies or features, and the repository does not contain multiple versions of packages.

3 Upgrade Optimization

The discussed complexity results provide convincing evidence that dependency solving is difficult to get right, more than developers might imagine at first. Several authors [39,34,40,55,51,54,24,36] have pointed out two main deficiencies of state-of-the-art package managers in the area of dependency solving—incompleteness and poor expressivity—some of them have proposed various alternative solutions.

A *dependency solving problem*, as usually faced by dependency solvers, can be described as consisting of: *(i)* a repository of all available packages (sometimes also referred to as a *package universe*); *(ii)* a subset of it denoting the set of currently installed packages on the target machine (*package status*); *(iii)* a *user request* usually asking to install, upgrade, or remove some packages. The expected output is a new package status that both is a proper installation (in the sense of Def. 3) and satisfies the user request. Note that, due to the presence of both implicit and explicit disjunctions in the dependency language, there are usually many valid solutions for a given upgrade problem. In fact, it has been shown in [4] that there are *exponentially* many solutions to upgrade problems in all non-trivial repositories.

A dependency solver is said to be *complete* if it is able to find a solution to an upgrade problem whenever one exists.

Given the huge amount of valid solutions to any given upgrade problem, we need languages that allow the user to express her preferences such as "favor solutions that minimize the amount of used disk space", "favor solutions that minimize the changes to the current package status", "do not install packages that are affected by outstanding security issues", etc.

Unfortunately, most state-of-the-art package managers are neither complete nor offer expressive user preference languages [53].

3.1 The Common Upgradeability Description Format

CUDF [52,4] (the Common Upgradeability Description Format)[2] is a language devised to solve the issues of completeness and expressivity by inducing a synergy among package managers developers and researchers in the various fields of constraint solving. At first glance, a CUDF document captures an instance of a

[2] `http://www.mancoosi.org/cudf/`, retrieved May 2013

```
preamble:
property: bugs: int = 0, suite: enum(stable,unstable) = "stable",

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true
...
request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

Fig. 3. Sample CUDF document

dependency solving problem using a human readable syntax, as shown in Fig. 3. CUDF is an extensible language—i.e. it allows to represent ad-hoc package properties that can then be used to express user preferences—and provides a formal semantics to unambiguously determine whether a given solution is correct with respect to the original upgrade problem or not.

CUDF is also neutral on both specific packaging and solving technologies. Several kinds of package manager-specific upgrade problems can be encoded in CUDF and then fed to solvers based on different constraint solving techniques. Fig. 4 enumerates a number of packaging technologies and solving techniques that can be used together, relying on CUDF for data exchange.

This is achieved by instrumenting existing package managers with the ability to communicate via the CUDF format with external dependency solvers. Such an arrangement, depicted in Fig. 5 and studied in [3,7], allows to share dependency solvers across package managers. Several modern package managers have followed this approach either offering the possibility to use external CUDF solvers as plugins, or even abandoning the idea of an integrated solver and always using external CUDF solvers. Examples are the APT package manager used by the Debian and Ubuntu distributions, the P2 provisioning platform for Eclipse plugins, and the OPAM package manager for the OCaml language.

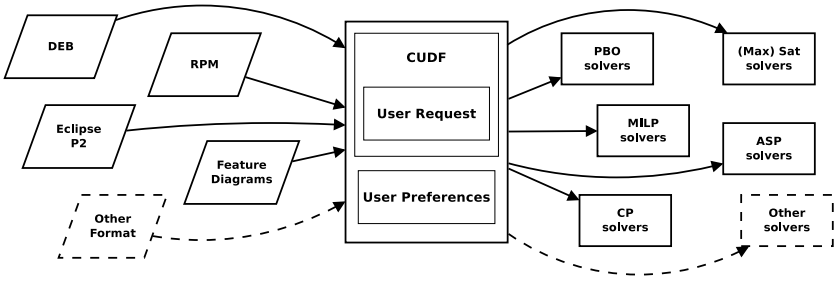


Fig. 4. Sharing upgrade problems and solvers among communities

3.2 User Preferences

In itself, CUDF does not mandate a specific language for user preferences, but *supports* them, in various ways. On one hand, CUDF captures and exposes all relevant characteristics of upgrade problems (e.g. package and user request properties) that are needed to capture user preferences in common scenarios [53]. Also, CUDF does so in an extensible way, so that properties that are specific to a given package technology can still be captured. On the other hand, the CUDF model is rigorous, providing a solid base to give a clear and measurable semantics to user preferences, which would allow to compare solutions and decide how well they score w.r.t. user preferences.

Several proposals of user preference languages have been advanced. The main challenge consists in finding a middle ground between the expressivity that users desire and the capabilities of modern constraint solvers.

Historically, OPIUM [55] has used SAT-based optimization to hard-code a fairly typical user preference, corresponding to the desire of minimizing the number of packages that are installed/removed to satisfy user request.

For the first time in [4], a flexible preference language has been proposed, based on a set of metrics that measure the distance between the original package status and the solution found by the dependency solver. Distance can be measured on various axes: the number of packages removed, newly installed, changed, that are not up to date (i.e. not at the latest available version), and with unsatisfied "weak" dependencies (i.e. packages that are "recommended" to be installed together with others, but not strictly required). Those metrics can be combined using a dictionary of aggregation functions that are commonly supported by solvers capable of multicriteria optimization [49], in particular lexicographic orderings and weighted sums. Using the resulting formalism it is possible to capture common user preference use cases such as the following *paranoid* one

$$paranoid = lex(-removed, -changed)$$

The solution scoring best under this criterion is the one with the smallest number of removed functionalities, and then with the smallest number of changes

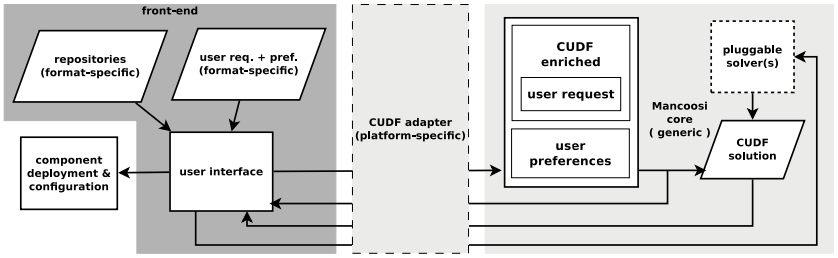


Fig. 5. Modular package manager architecture

(e.g. upgrade/downgrade/install actions). A *trendy* preference, i.e. the desire of having the most recent versions of packages, is also easy to write as:

$$trendy = lex(-removed, -notuptodate, -unsatrec, -new)$$

This set of preference combinators is bound to grow to encompass new user needs. For example, it is often the case that a single source package can produce many binary packages, and that using a mix of binary packages coming from different versions of the same source package is problematic. In recent work, it has been shown how to implement an optimization criterion that allows to specify that some packages need to be *aligned*, for different notions of alignment [23].

3.3 The MISC Competition

The existence of a language like CUDF allows to assemble a corpus of challenging (for existing dependency solvers) upgrade problems coming from actual users of different package managers. Using such a corpus, researchers have established the MISC (for Mancoosi International Solver Competition)[3] that has been run yearly since 2010. The goal of the competition is to advance the state of the art of real dependency solvers, similarly to what has happened in others fields with, e.g., the SAT competition [35].

A dozen solvers have participated in the various editions, attacking CUDF-encoded upgrade problems using solvers based on a wide range of constraint solving techniques. Table 2 shows a sample of MISC participants from the 2010 and 2011 editions.

Analysis of the competition results has allowed us to experimentally establish the limits of state-of-the-art solvers. In particular, they have been shown to significantly degrade their ability to (quickly) find solutions as the number of used package repositories grows, which is a fairly common use case. Each competition edition has established one or more winners, e.g. one winner in the *trendy* track and one in the *paranoid* one. Modular package managers that follow the architecture of Fig. 5 can then use winning solvers, or other entrants, as their dependency solver of choice.

[3] `http://www.mancoosi.org/misc/`, retrieved May 2013

Table 2. Sample of MISC competition entrants, ed. 2010 and 2011

solver	author/affiliation	technique/solver
apt-pbo [54]	Trezentos / Caixa Magica	Pseudo Boolean Optimization
aspcud	Matheis / University of Potsdam	Answer Set Programming
inesc [9]	Lynce et. al / INESC-ID	Max-SAT
p2cudf [9]	Le Berre and Rapicault / Univ. Artois	Pseudo Boolean Optimization / Sat4j (`www.sat4j.org`)
ucl	Gutierrez et al. / Univ. Luvain	Graph constraints
unsa [44]	Michel et. al / Univ. Sophia-Antipolis	Mixed Integer Linear Programming / CPLEX (`www.cplex.com`)

Solvers able to handle these optimization combinators can also be used for a variety of other purposes. It is worth mentioning one of the most unusual, which is building minimum footprint virtual images for the cloud: as noticed in [46], virtual machine images often contain largely redundant package selections, wasting disk space in cloud infrastructures. Using the toolchain available in the `dose` library,[4] which is at the core of the MISC competition infrastructure, one can compute the smallest distribution containing a given set of packages. This problem has actually been used as one of the track of the 2012 edition of the MISC competition.

More details on CUDF and MISC are discussed in [4,7].

4 Quality Assurance of Component Repositories

A particularly fruitful research line tries to solve the problems faced by the maintainers of component repositories, and in particular of FOSS distributions.

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into and the removal of packages from it. With the package count in the tens of thousands (over 35.000 in the latest Debian development branch as of this writing), there is a serious need for tools that help answering *efficiently* several different questions. Some are related to the current state of a distribution, like: "What are the packages that cannot be installed (i.e., that are *broken*) using the distribution I am releasing?", "what are the packages that block the installation of many other packages?", "what are the packages most depended upon?". Other questions concern more the evolution of a distribution over time, like: "what are the *broken* packages that can only be fixed by changing them (as opposed to packages they depend on)?", "what are the *future version changes* that will break the most packages in the distribution?", "are there sets of packages that were installable together in the previous release, and can no longer be installed together in the next one?".

In this section, we highlight the most significant results obtained over the past years that allow to answer some of these questions, and led to the development of tools which currently are being adopted by distribution maintainers.

[4] `http://www.mancoosi.org/software/`, retrieved May 2013

4.1 Identifying Broken Packages

As we have seen in Sect. 2.3, the problem of determining whether a single package is installable using packages from a given repository is NP-hard. Despite this limiting result, modern SAT solvers are able to handle easily the instances coming from real world repositories. This can be explained by observing that explicit conflicts between packages are not very frequent, even if they are crucial when they exist, and that when checking installability in a single repository one usually finds only one version per package, hence no implicit conflicts. As a consequence, there is now a series of tools, all based on the original `edos-debcheck` tool developed by Jérôme Vouillon in 2006 [43], that are part of the `dose` library and can check installability of Debian or RPM packages, as well as Eclipse plugins, in a very short time: a few seconds on commodity desktop hardware are enough to handle the ≈ 35.000 packages from the latest Debian distribution.[5]

4.2 Analyzing the Dependency Structure of a Repository

Identifying the packages that are not installable in a repository is only the first basic analysis which is of interest for a distribution maintainer: among the large majority of packages that are installable, not all have the same importance, and not all can be installed together.

It is quite tempting, for example, to use the number of incoming dependencies on a package as a measure of its importance. Similarly, it is tempting to analyze the dependency graph trying to identify "weak points" in it, along the tradition of studies on small-world networks [8]. Several studies in the literature do use explicit dependencies, or their transitive closure, to similar ends (e.g. [37,42]). The explicit, syntactic dependency relation $p \rightarrow q$ is however too imprecise for them and can be misleading in many circumstances. Intuitively, this is so because paths in the explicit dependency graph might connect packages that are incompatible, in the sense that they cannot be installed together. To solve that problem we need to distinguish between the syntactic dependency graph and a more meaningful version of it that takes into account the actual semantics of dependencies and conflicts.

This was the motivation for introducing the notion of *strong dependency* [1] to identify the packages that are at the core of a distribution.

Definition 8. *A package p strongly depends on q (written $p \Rightarrow q$) with respect to a repository R if it is not possible to install p without also installing q.*

This property is easily seen equivalent to the implication $p \rightarrow q$ in the logical theory obtained by encoding the repository R, so in the general case this problem is co-NP-complete, as it is the dual of an installation problem, and the strong dependency graph can be huge, because it is transitive. Nevertheless, it is possible on practical instances to compute the strong dependency graph of a recent

[5] A daily updated showcase of uninstallable Debian packages, used by the distribution for quality assurance purposes, is currently available at `http://edos.debian.net/edos-debcheck/` (retrieved January 2013).

Table 3. Top sensitive packages in Debian 5.0 "Lenny"

#	package	deps	**s. deps**	closure
1	`gcc-4.3-base`	43	**20128**	20132
2	`libgcc1`	3011	**20126**	20130
3	`libc6`	10442	**20126**	20130
4	`libstdc++6`	2786	**14964**	15259
5	`libselinux1`	50	**14121**	14634
6	`lzma`	4	**13534**	13990
7	`libattr1`	110	**13489**	14024
8	`libacl1`	113	**13467**	14003
9	`coreutils`	17	**13454**	13991
10	`dpkg`	55	**13450**	13987
11	`perl-base`	299	**13310**	13959
12	`debconf`	1512	**11387**	12083
13	`libncurses5`	572	**11017**	13466
14	`zlib1g`	1640	**10945**	13734
15	`libdb4.6`	103	**9640**	13991
	...			

Debian distribution in a few hours on a modern multicore machine. The optimized algorithms able to do so have been discussed in [1] and are implemented in the `dose` toolchain.[6]

Once the strong dependencies graph is known, it is possible to define the *impact set* of a package, as the set of packages that strongly depend on it: this is a notion of robustness, as removing p from the distribution renders uninstallable all packages in its impact set, while this is not the case if one uses direct or transitive dependencies.

In Table 3 are shown the ten packages from the Debian Lenny distribution with the largest impact set, and it is easy to see that the number of direct incoming dependencies is totally unrelated to the real importance of the package, while the number of transitive incoming dependencies is always an overapproximation.

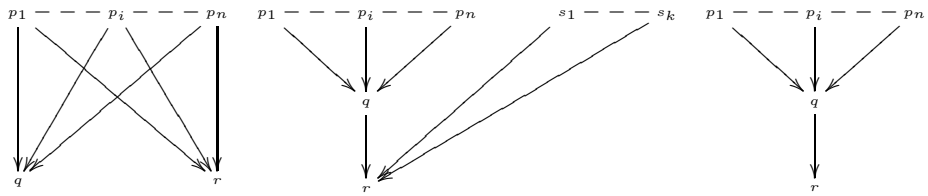
In the list of Table 3, a knowledgable maintainer will recognize a cluster of interrelated packages: `gcc-4.3-base`, `libgcc1` and `libc6` are all essential components of the C library, and they have similar sized impact sets. In the general case, though, as shown by the examples configurations drawn in Fig. 6, two packages having similar sized impact sets need not be correlated.

To identify those packages that are correlated, and identify the most relevant ones among them, one can define, on top of the strong dependency graph, a dominance relation similar to the one used in traditional flow graphs [41].

Definition 9 (Strong Dominance). *We say that q strongly dominates r if:*

- *q strongly depends on r, and*
- *every package that strongly depends on r also strongly depends on q.*

[6] `http://www.mancoosi.org/software/`, retrieved May 2013



(a) Coincidence (b) General case (c) Dominance

Fig. 6. Significant configurations in the strong dependency graph

Intuitively, in a strong dominance configuration, that looks like Fig. 6c, the strong dependency on r of packages in its impact set is "explained by" their strong dependency on q.

Strong dominance can be computed efficiently [13] and properly identifies many relevant clusters of packages related by strong dependencies like the `libc6` one, but the tools built on the work of [1] also allow to capture partial dominance situations as in Fig. 6b.

4.3 Analyzing the Conflict Structure of a Repository

In a repository there are packages that can be installed in isolation, but not together: despite the existence of interesting approaches that make it possible to reduce the need for package conflicts when installing user-level packages [29], there are good reasons for making sure, for example, that only one mail transport agent, or database server, is installed on a given machine, and that only one copy of a dynamic library needs to be updated if a security issue is uncovered. This is why in the current Debian distribution one can find over one thousand explicit conflicts declared among packages [10,11].

Once conflicts are part of a distribution, it is important to be able to assess their impact, identifying those packages that are incompatible. This is not an easy task, even when looking just for incompatibilities between package pairs, that are known as *strong conflicts* [17], as opposed to "ordinary" conflicts.

Definition 10 (Strong Conflicts). *The packages in S are in* strong conflict *if they can never be installed all together.*

Indeed, by duality with the installability problem, one obtains the following

Theorem 2. *Determining whether S is in strong conflict in a repository R is co-NP-complete.*

In practice, though, strong conflicts can be computed quite efficiently [25], and this allows to identify packages with an abnormal number of incompatibilities, that are simply undetectable using other kinds of metrics. For example, Table 4 lists the ten packages from Debian Lenny with the highest number of strong

conflicts, and the package `ppmtofb` clearly stands out as a problematic one: it is installable, so it will not be flagged by the `edos-debcheck` tool, but it is in practice incompatible with a large part of the Debian distribution. It turned out that this package depended on an old version of Python, which was phased out, but was never updated; after reporting the issue, the package was dropped from the distribution, because of lack of maintainers, but could have been adapted to the latest Python versions too.

Table 4. Top packages with the highest number of strong conflicts in Debian Lenny

Strong Conflicts	Package	Explicit Conflicts	Explicit Dependencies	Cone Size	Cone Height
2368	ppmtofb	2	3	6	4
127	libgd2-noxpm	4	6	8	4
127	libgd2-noxpm-dev	2	5	15	5
107	heimdal-dev	2	8	121	10
71	dtc-postfix-courier	2	22	348	8
71	dtc-toaster	0	11	429	9
70	citadel-mta	1	6	123	9
69	citadel-suite	0	5	133	9
66	xmail	4	6	105	8
63	apache2-mpm-event	2	5	122	10

More generally, one is interested in identifying the sets of packages that are incompatible, and in providing a way for a distribution maintainer to visualize the problematic configurations. With over 35.000 packages, and hundreds of thousands of relationships, this may look like an impossible task.

The key idea for properly addressing this challenge is to extract from the original, huge repository, a much smaller one, called a *coinstallability kernel*, that contains a representative of each package of the original repository, and preserves co-installability of packages [25]. That is, even if a coinstallability kernel is a much more compact representation of package relationships than the original one, all relevant information to decide whether packages are co-installable or not is retained by it.

To obtain a coinstallability kernel, we start from the original repository and perform a series of transformations on it. As a first step, one builds a transitive closure of the dependency relation, reminiscent of the conversion to conjunctive normal form of propositional formulae, but dropping at the same time some redundant dependencies to avoid combinatorial explosion. This phase produces a repository that has a two-level structure, which one may simplify further by removing other redundant dependencies that are exposed by the transitivization phase; after closing the dependency function reflexively, one can finally collapse packages that have the same behavior with respect to co-installability into equivalence classes, and then remove the reflexive dependencies to obtain a compact visualization of the kernel of the repository.

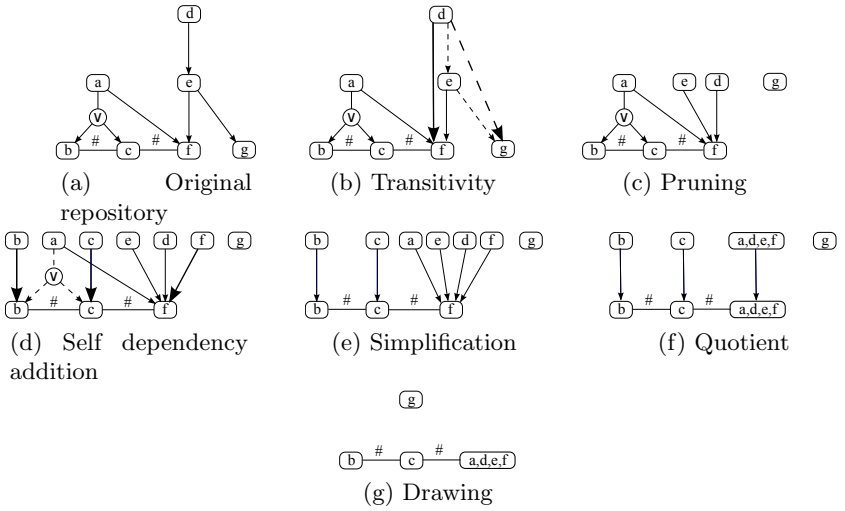


Fig. 7. Transformations of a repository. Conflicts edges are denoted with #; arrows denote direct (conjunctive) dependencies, whereas disjunctive dependencies use explicit "OR" nodes. Dashed lines are used, at each phase, to highlight edges that will disappear in the next phase.

These phases are shown on a sample repository in Fig. 7: it is clear from this example that in the final repository package `g` can always be installed, while `b` and `c` are incompatible, and all the packages `a, d, e, f` behave the same with respect to co-installability, and are only incompatible with `c`. Due to the fundamental theorems proved, and machine checked, in [25], this is also the case in the original repository.

On real-world repositories, the simplification obtained is astonishing, as can be seen in the following table, that also indicates the running time of the `coinst` tool[7] on a commodity laptop:

	Debian		Ubuntu		Mandriva	
	before	after	before	after	before	after
Packages	28919	1038	7277	100	7601	84
Dependencies	124246	619	31069	29	38599	8
Conflicts	1146	985	82	60	78	62
Running time (s)		10.6		1.19		11.6

4.4 Predicting Evolutions

Some aspects of the quality assessment in FOSS distributions are best modeled by using the notion of *futures* [2,5] of a package repository. This allows to investigate under which conditions a potential future problem may occur, or what

[7] `http://coinst.irill.org`, retrieved January 2013

```
Package: bar
Version: 2.3

Package: baz
Version: 2.5
Conflicts: bar (> 2.4)
```

```
Package: foo
Version: 1
Depends: (baz (=2.5) | bar (=2.3)),
         (baz (<2.3) | bar (>2.6))
```

Fig. 8. Package `foo` in version 1 is outdated

changes to a repository are necessary to make a currently occurring problem go away. This analysis can give package maintainers important hints about how problems may be solved, or how future problems may be avoided. The precise definition of these properties relies on the definition of the possible future of a repository:

Definition 11 (Future). *A repository F is a* future *of a repository R if the following two properties hold:*

uniqueness *$R \cup F$ is a repository; this ensures that if F contains a package p with same version and name as a package q already present in R, then $p = q$;*

monotonicity *For all $p \in R$ and $q \in F$: if $p.n = q.n$ then $p.v \leq q.v$.*

In other words, when going from the current repository to some future of it one may upgrade current versions of packages to newer versions, but not downgrade them to older versions (monotonicity). One is not allowed to change the meta-data of a package without increasing its version number (uniqueness), but besides this the upgrade may modify the meta-data of a package in any possible way, and may even remove a package completely from the repository, or introduce new packages. This notion models all the changes that are possible in the maintenance process usually used by distribution editors, even if the extreme case of a complete change of meta-data allowed in this model is quite rare in practice. Note that the notion of future is not transitive as one might remove a package and then reintroduce it later with a lower version number.

The first property using futures that we are interested in is the following one:

Definition 12 (Outdated). *Let R be a repository. A package $p \in R$ is* outdated *in R if p is not installable in any future F of R.*

That is, p is outdated in R if it is not installable (since R is itself one of its futures) and if it has to be upgraded to make it ever installable again. In other words, the only way to make p installable is to upload a fixed version of the package since no modification to other packages than p can make p installable. This information is useful for quality assurance since it pinpoints packages where action is required. An example of an outdated package is given in Fig. 8.

Definition 13 (Challenges). *Let R be a repository, $p, q \in R$, and q installable in R. The pair $(p.n, v)$, where $v > p.v$,* challenges *q if q is not installable in any future F which is obtained by upgrading p to version v.*

Intuitively $(p.n, v)$ challenges q, when upgrading p to a new version v without touching any other package makes q not installable. This permits to pinpoint critical future upgrades that challenge many packages and that might therefore need special attention before being pushed to the repository. An example is given in Fig. 9.

```
Package: foo                                  Package: baz
Version: 1.0                                  Version: 1.0
Depends: bar (<= 3.0) | bar (>= 5.0)          Depends: foo (>= 1.0)

Package: bar
Version: 1.0
```

Fig. 9. Package `bar` challenges package `foo` for versions in the interval $]3.0, 5.0[$

The problem in deciding these properties is that any repository has an infinite number of possible futures. The two properties we are interested in belong to the class of so-called *straight* properties. For this class of properties it is in fact sufficient to look at a finite set of futures only which cover all of the problems that may occur in any future. One can show [5] that it is sufficient to look at futures where no package has been removed and new packages have been introduced only when their name was already mentioned in R, and where all new versions of packages have no conflicts and no dependencies. For any package there is an infinite space of all future version numbers, however, there is only a finite number of equivalence classes of these with respect to observational equivalence where the observations are the constraints on versions numbers used in R.

In reality, the definition of a future is more involved than the one given in Def. 11. In almost all distributions, packages are in fact not uploaded independently from each other but are updated together with all other packages stemming from the same *source package*. The complete definition of a future also takes into account a notion of *clusters* of packages, which are in our case formed by all binary packages stemming from the same source. Def. 13 has to be adapted accordingly, by allowing for all packages in the same cluster as p to be upgraded.

The full version of the algorithms in presence of package clusters, together with their proof of soundness, can be found in [5].

The top challenging upgrades in Debian Lenny found by our tool are listed in Table 5. Regularly updated reports on outdated Debian packages are available as part of the distribution quality assurance infrastructure.[8]

[8] `http://edos.debian.net/outdated.php`, retrieved January 2013

Table 5. Top 13 challenging upgrades in Debian lenny

Source	Version	Target Version	Breaks
python-defaults	2.5.2-3	≥ 3	1079
python-defaults	2.5.2-3	$2.6 \leq . < 3$	1075
e2fsprogs	1.41.3-1	any	139
ghc6	6.8.2dfsg1-1	$\geq$ 6.8.2+	136
libio-compress-base-perl	2.012-1	$\geq$ 2.012.	80
libcompress-raw-zlib-perl	2.012-1	$\geq$ 2.012.	80
libio-compress-zlib-perl	2.012-1	$\geq$ 2.012.	79
icedove	2.0.0.19-1	$>$ 2.1-0	78
iceweasel	3.0.6-1	$>$ 3.1	70
haskell-mtl	1.1.0.0-2	$\geq$ 1.1.0.0+	48
sip4-qt3	4.7.6-1	$>$ 4.8	47
ghc6	6.8.2dfsg1-1	6.8.2dfsg1+ $\leq . <$ 6.8.2+	36
haskell-parsec	2.1.0.0-2	$\geq$ 2.1.0.0+	29

With the same philosophy of identifying the impact of repository evolutions, it is important for quality assurance to be able to spot easily whether a new release has introduced new errors, and one particular error that affects FOSS distributions is the introduction of new incompatibilities among packages that were co-installable in a previous version. At first sight, identify such errors seems unfeasible: one would need to enumerate all possible sets of incompatible packages in the new distribution, and then check whether they were already incompatible in the previous release. Since there are 2^n package sets in a distribution with n packages, and n is in the tens of thousands, this approach is unfeasible. Recent work has shown that by introducing a notion of *cover* for incompatible package sets, it is actually possible to identify all such new errors in a very limited amount of time [21].

5 Related Work

Upgrade Simulation. Incompleteness and poor expressivity are just some of the issues that might be encountered while upgrading FOSS-based systems [18,24]. Several other issues can be encountered during actual package deployment, due to the unpredictability of *configuration scripts* execution on target machines. The formal treatment of those scripts is particularly challenging due to the fact that they are usually implemented in languages such as shell script and Perl, which are Turing-complete languages that also heavily rely on dynamic features such as shell expansions.

Model-driven techniques [12] have been applied to first capture the syntax and semantics of common high-level actions performed by configuration scripts in popular FOSS distributions, and then to instrument package deployment with simulators able to predict a significant range of upgrade failures before the actual target machine is affected [22,48,14,28].

Packages and Software Components. Packages share important features with *software components* [50,38], but exhibit also some important differences. On the one hand, packages, like components, are reusable software units which can be combined freely by a system administrator; they are also independent units that follow their own development time-line and versioning scheme.

On the other hand, packages, unlike what happens in many software component models, cannot be composed to build a larger component, and it is not possible to install more than one copy of a given package on a given system. Furthermore, installation of packages, and execution of software contained in packages, acts on shared resources that are provided by the operating system, like creating files on the file system, or interacting through the systems input/output devices. As a consequence, packages may be in conflict with each other, a phenomenon which is not (yet?) commonplace for software components.

Software components come with an interface describing their required and provided services. In the case of packages, requirements and provided features are given by symbolic names (either names of packages, or names of abstract features) whose semantics is defined separately from the package model. For instance, a policy document may describe how an executable must behave in order to provide a feature `mail-transport-agent`, or an external table will tell us which symbols have been provided in version 1.2.3 of library `libfoo`.

Packages and Software Product Lines. Issues similar to dependency solving are faced by semi-automatic configurators for software product lines [16]: they too have dependencies and conflicts, this time among features, and need to find a subset of them that work well together. Independently from packaging work, the application of SAT solving to feature selection has been investigated, among others, in [34].

The analogy between software product lines (SPL) and package repositories have been recently observed in other works, that explicitly show how to map one formalism into the other and vice-versa. The goal is to allow sharing of tools and techniques between the two worlds. The mapping from software product lines, based on the feature diagram formalism, to package repositories has been established in [26]; whereas a converse mapping, from Debian packages to SPL, has been more recently proposed in [32].

Packages and the Cloud. The idea of relying on automated tools to configure a software system based on (i) a repository of components and (ii) a user request to satisfy, can be applied in contexts larger than a single machine. The idea can in fact be extended to networks of heterogeneous machines, where each machine is associated to a specific package repository, and to higher-level services that span multiple machines and might induce inter-dependencies (and conflicts) among them.

This approach has been recently explored in the context of the Aeolus project [27], which directly tries to apply constraint solving to network and cloud settings and, with a slightly narrower but more easily automatable scope, also in the context of the Engage system [31].

References

1. Abate, P., Boender, J., Di Cosmo, R., Zacchiroli, S.: Strong dependencies between software components. In: ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 89–99 (2009)
2. Abate, P., Di Cosmo, R.: Predicting upgrade failures using dependency analysis. In: Abiteboul, S., Böhm, K., Koch, C., Tan, K.L. (eds.) Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, Hannover, Germany, April 11-16, pp. 145–150. IEEE (2011)
3. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Mpm: a modular package manager. In: CBSE 2011: 14th International ACM SIGSOFT Symposium on Component Based Software Engineering, pp. 179–188. ACM (2011)
4. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Dependency solving: a separate concern in component evolution management. Journal of Systems and Software 85(10), 2228–2240 (2012)
5. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Learning from the future of component repositories. In: CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering, pp. 51–60. ACM (2012)
6. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Learning from the future of component repositories. Science of Computer Programming (2012) (to appear)
7. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: A modular package manager architecture. Information and Software Technology 55(2), 459–474 (2013)
8. Albert, R., Jeong, H., Barabási, A.L.: Error and attack tolerance of complex networks. Nature 406(6794), 378–382 (2000)
9. Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P.: Solving Linux upgradeability problems using boolean optimization. In: LoCoCo: Logics for Component Configuration. EPTCS, vol. 29, pp. 11–22 (2010)
10. Artho, C.V., Di Cosmo, R., Suzaki, K., Zacchiroli, S.: Sources of inter-package conflicts in debian. In: LoCoCo 2011 International Workshop on Logics for Component Configuration (2011)
11. Artho, C.V., Suzaki, K., Di Cosmo, R., Treinen, R., Zacchiroli, S.: Why do software packages conflict? In: MSR 2012: 9th IEEE Working Conference on Mining Software Repositories, pp. 141–150. IEEE (2012)
12. Bézivin, J.: On the unification power of models. SOSYM 4(2), 171–188 (2005)
13. Boender, J.: Efficient computation of dominance in component systems (Short paper). In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 399–406. Springer, Heidelberg (2011)
14. Cicchetti, A., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: A model driven approach to upgrade package-based software systems. In: Maciaszek, L.A., González-Pérez, C., Jablonski, S. (eds.) ENASE 2008/2009. CCIS, vol. 69, pp. 262–276. Springer, Heidelberg (2010)
15. Clayberg, E., Rubel, D.: Eclipse Plug-ins, 3rd edn. Addison-Wesley Professional (December 2008)
16. Clements, P., Northrop, L.: Software product lines. Addison-Wesley (2002)
17. Cosmo, R.D., Boender, J.: Using strong conflicts to detect quality issues in component-based complex systems. In: Padmanabhuni, S., Aggarwal, S.K., Bellur, U. (eds.) ISEC, pp. 163–172. ACM (2010)
18. Crameri, O., Knezevic, N., Kostic, D., Bianchini, R., Zwaenepoel, W.: Staged deployment in mirage, an integrated software upgrade testing and distribution system. SIGOPS Oper. Syst. Rev. 41(6), 221–236 (2007)

19. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
20. Des Rivières, J., Wiegand, J.: Eclipse: a platform for integrating development tools. IBM Systems 43(2), 371–383 (2004)
21. Vouillon, J., Di Cosmo, R.: Broken sets in software repository evolution. In: ICSE 2013. ACM (to appear, 2013)
22. Di Cosmo, R., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Supporting software evolution in component-based FOSS systems. Science of Computer Programming 76(12), 1144–1160 (2011)
23. Di Cosmo, R., Lhomme, O., Michel, C.: Aligning component upgrades. In: Drescher, C., Lynce, I., Treinen, R. (eds.) LoCoCo 2011 International Workshop on Logics for Component Configuration, vol. 65, pp. 1–11 (2011)
24. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWUp 2008: Hot Topics in Software Upgrades. ACM (2008)
25. Di Cosmo, R., Vouillon, J.: On software component co-installability. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT FSE, pp. 256–266. ACM (2011)
26. Di Cosmo, R., Zacchiroli, S.: Feature diagrams as package dependencies. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 476–480. Springer, Heidelberg (2010)
27. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012)
28. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards maintainer script modernization in FOSS distributions. In: IWOCE 2009: International Workshop on Open Component Ecosystem, pp. 11–20. ACM (2009)
29. Dolstra, E., Löh, A.: NixOS: A purely functional linux distribution. In: ICFP (2008) (to appear)
30. EDOS Project: Report on formal management of software dependencies. EDOS Project Deliverables D2.1 and D2.2 (March 2006)
31. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI, pp. 263–274. ACM (2012)
32. Galindo, J., Benavides, D., Segura, S.: Debian packages repositories as software product line models. towards automated analysis. In: Dhungana, D., Rabiser, R., Seyff, N., Botterweck, G. (eds.) ACoTA. CEUR Workshop Proceedings, vol. 688, pp. 29–34. CEUR-WS.org (2010)
33. Gonzalez-Barahona, J., Robles, G., Michlmayr, M., Amor, J., German, D.: Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14(3), 262–285 (2009)
34. Janota, M.: Do SAT solvers make good configurators? In: SPLC: Software Product Lines Conference, vol. 2, pp. 191–195 (2008)
35. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine 33(1) (2012)
36. Jenson, G., Dietrich, J., Guesgen, H.W.: An empirical study of the component dependency resolution search space. In: Grunske, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 182–199. Springer, Heidelberg (2010)
37. LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. CoRR cs.SE/0411096 (2004)
38. Lau, K.K., Wang, Z.: Software component models. IEEE Trans. Software Eng. 33(10), 709–724 (2007)

39. Le Berre, D., Parrain, A.: On SAT technologies for dependency management and beyond. In: SPLC 2008: Software Product Lines Conference, vol. 2, pp. 197–200 (2008)
40. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem. In: IWOCE 2009: International Workshop on Open Component Ecosystems, pp. 21–30. ACM (2009)
41. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
42. Maillart, T., Sornette, D., Spaeth, S., von Krogh, G.: Empirical tests of zipf's law mechanism in open source linux distribution. Phys. Rev. Lett. 101, 218701 (2008), http://link.aps.org/doi/10.1103/PhysRevLett.101.218701
43. Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: ASE 2006: Automated Software Engineering, pp. 199–208. IEEE (2006)
44. Michel, C., Rueher, M.: Handling software upgradeability problems with MILP solvers. In: LoCoCo 2010: Logics for Component Configuration. EPTCS, vol. 29, pp. 1–10 (2010)
45. OSGi Alliance: OSGi Service Platform, Release 3. IOS Press, Inc. (2003)
46. Quinton, C., Rouvoy, R., Duchien, L.: Leveraging feature models to configure virtual appliances. In: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, CloudCP 2012, pp. 2:1–2:6. ACM, New York (2012), http://doi.acm.org/10.1145/2168697.2168699
47. Raymond, E.S.: The cathedral and the bazaar. O'Reilly (2001)
48. Ruscio, D.D., Pelliccione, P., Pierantonio, A.: EVOSS: A tool for managing the evolution of free and open source software systems. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE, pp. 1415–1418. IEEE (2012)
49. Steuer, R.E.: Multiple Criteria Optimization: Theory, Computation and Application. Wiley (1986)
50. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. Addison-Wesley (1998)
51. Treinen, R., Zacchiroli, S.: Solving package dependencies: from EDOS to Mancoosi (2008)
52. Treinen, R., Zacchiroli, S.: Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project (November 2009), http://www.mancoosi.org/reports/tr3.pdf
53. Treinen, R., Zacchiroli, S.: Expressing advanced user preferences in component installation. In: IWOCE 2009: International Workshop on Open Component Ecosystems, pp. 31–40. ACM (2009)
54. Trezentos, P., Lynce, I., Oliveira, A.: Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization. In: ASE 2010: Automated Software Engineering, pp. 427–436. ACM (2010)
55. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: OPIUM: Optimal package install/uninstall manager. In: ICSE 2007: International Conference on Software Engineering, pp. 178–188. IEEE (2007)