A Modular Package Manager Architecture[☆]

Pietro Abate[a], Roberto Di Cosmo[a], Ralf Treinen[a], Stefano Zacchiroli[a]

[a] *Univ Paris Diderot, Sorbonne Paris Cité*
PPS, UMR 7126, CNRS, F-75205 Paris, France

Abstract

The success of modern software distributions in the Free and Open Source world can be explained, among other factors, by the availability of a large collection of software packages and the possibility to easily install and remove those components using state-of-the-art *package managers*. However, package managers are often built using a monolithic architecture and hard-wired and ad-hoc *dependency solvers* implementing some customized heuristics.

In this paper we propose a modular architecture relying on precise interface formalisms that allows the system administrator to choose from a variety of dependency solvers and backends. We argue that this is the path that leads to the next generation of package managers that will deliver better results, offer more expressive preference languages, and be easily adaptable to new platforms.

We have built a working prototype—called MPM—following the design advocated in this paper, and we show how it largely outperforms a variety of current package managers.

Keywords:
software dependencies, software repositories, software components, package manager, open source
2010 MSC: 68N30, 68M99

1. Introduction

Free and Open Source Software (FOSS) distributions, as well as other complex software platforms, strive to provide modular software components, called *packages*, that can be assembled to provide the user with the desired functionalities. Packages are equipped with a rich set of metadata providing information

[☆]This work has been partially performed at IRILL www.irill.org

Email addresses: abate@pps.jussieu.fr (Pietro Abate), roberto@dicosmo.org (Roberto Di Cosmo), treinen@pps.univ-paris-diderot.fr (Ralf Treinen), zack@pps.univ-paris-diderot.fr (Stefano Zacchiroli)

URL: http://www.pps.jussieu.fr/~abate (Pietro Abate), http://www.dicosmo.org (Roberto Di Cosmo), http://www.pps.univ-paris-diderot.fr/~treinen/ (Ralf Treinen), http://upsilon.cc/~zack (Stefano Zacchiroli)

on their content and the relationships to other packages, that describe the requirements for a package to run properly on a target system.

Packages, as found in FOSS distributions, share important features with *software component models* [14], but exhibit also some important differences. On one side, 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 side, 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*).

A key component for maintaining and deploying software systems based on packages are the tools used to perform installation, upgrade and removal of packages on the target machines. These tools, called *package managers*, incorporate numerous functionalities: they allow to retrieve components from remote repositories, and eventually checking their integrity; they compute upgrade paths that respect inter-component constraints (a functionality known as *dependency solving*); they handle the interaction with the user to allow for fine-tuning of the choice of components; and finally, they perform the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered.

Package managers take a very abstract view by considering only constraints between packages identified by names. Even though the package model is quite simple and abstract, package managers face two major challenges:

Logical complexity Packages are defined in terms of positive (dependencies) and negative constraints (conflicts), and dependencies may be composed by using logical conjunctions and disjunctions.

Scale Package repositories include tens of thousands of packages. This challenge is even more complex when package managers may pick packages from several repositories.

Until recently, package managers in FOSS distributions followed a monolithic architecture (re-)implementing all functionalities to fit specific formats of

metadata and user requests. In particular, dependency solving was often implemented by ad-hoc algorithms instead of employing well known solver technologies. Surprisingly little was known about the intrinsic complexity of dependency solving. It is only in [8] that some of the authors have shown that for packages in FOSS distributions determining whether a component can be installed is an NP-complete problem. This result has been established by showing the equivalence of package installation with Boolean satisfiability, which has opened the door to show that installation in other component models is NP-complete as well. These results and the straightforward encoding into Boolean satisfiability [18] have pushed various communities to incorporate SAT solvers directly in package managers, instead of writing ad-hoc solvers as it was previously the case [16, 21, 26, 27].

In this paper—which extends and formalizes the preliminary results of [2]—we argue that decoupling dependency solving from other functionalities will yield better package managers

- that succeed in finding an upgrade path where existing package managers fail,
- that are more powerful by accepting an input language that is more expressive than the ones currently supported,
- and that are more flexible by being easily adaptable to new platforms.

We propose a modular architecture to build component managers that decouples the front-end, which is in charge of interacting with the user and installing and removing individual components, from a generic back-end, which is in charge of finding the best upgrade path according to some user-specified criteria. As a uniform interface between the front-end and the back-end, our architecture relies on two domain specific languages: the Common Upgradeability Description Format (CUDF), which captures all the relevant information about component dependencies, and the user preferences language, which describes the criteria used to determine the best solution.

In particular we describe MPM, the *Mancoosi Package Manager*, which is a proof-of-concept implementation of this modular package manager architecture for Debian based systems. MPM largely outperforms the mainstream package managers available in the Debian FOSS distribution in terms of quality of the proposed solution.

This article is organized as follows: Section 2 introduces the package installation problem, describes the state of the art in the area of package managers, and provides a paradigmatic example of the limitations of current tools. Section 3 presents the modular architecture that we advocate for building package managers, and formally defines the two interface languages used to interconnect their components, CUDF and the user preferences language. Section 4 introduces MPM, our new modular package manager which is able to cope efficiently with different installation scenarios. Section 5 gives an overview of the performances of MPM in comparison with other package manages. Before concluding, we discuss related and future work in Section 6.

The appendix contains the precise syntax (Appendix A) and semantics (Appendix B) of the CUDF format, as well as our translation from Debian metadata and RPM meta-data to CUDF (Appendix C and Appendix D).

2. The upgrade problem

Mainstream FOSS distributions undergo a quality assurance process which aims, among other goals, at assuring a high degree of coherence of the packages contained in the distribution. In particular, a stable distribution will avoid shipping packages referring to other packages not included in the same distribution, and excludes packages which are impossible to install because of some unsatisfiable relation to other packages in the same distribution [23]. Furthermore, a released FOSS distribution usually contains only one version of each package.[1] As a consequence, the initial installation of a FOSS system from scratch usually runs little risk of incurring dependency problems, and installing, removing, and upgrading components in such a scenario is a task that can be correctly handled by the large majority of legacy package managers.

However, as new releases are rolled out, it is common practice to add the new stable distribution as an extra source of packages (or package "baseline") and use it to perform upgrades on the machines, instead of reinstalling the machine from scratch, or performing a full upgrade. Unfortunately, as we will show in Section 5, using multiple package sources makes the problem considerably harder because there are now multiple versions of packages from which to choose, and the set of possible configurations of the machine grows very quickly.

As time goes by, this situation often leads to what is known as *dependency hell*: the user gets entangled in an inextricable web of dependencies and conflicts that state-of-the-art package managers are unable to handle. Users are left on their own, and as a last resort the user may even be requested to guide interactively the package manager, by suggesting choices among packages which may seem totally unrelated to the original request.

2.1. A paradigmatic example

As a real-world example of this unsatisfactory situation, we have taken a stable installation of the Debian `sarge` release, gradually added some more recent releases, `etch` and `lenny`, and looked at how difficult it is to install a single, apparently innocuous package, `python-simpy`, that provides a process-based discrete-event simulation language based on Python. This configuration is far from unusual since `sarge`, `etch` and `lenny` are three consecutive stable Debian releases, which have passed strict quality assurance tests.

In this particular configuration, three versions of the `python-simpy` package are available: `1.5.1-2`, `1.7.1-1` and `1.8.1`.

[1] Notwithstanding the case where different upstream versions of the same software are organized into packages with distinct names, as it is often the case for packages of the Linux kernel

We tested the package managers `apt-get` (v. 0.9.7), `aptitude` (v. 0.6.7), `smart` (v. 1.4), and `cupt` (v. 2.5.6); their default behavior is to try to install the latest available version of the requested package (`python-simpy`), which is `1.8.1`. The results obtained in our tests (see Section 5) are not encouraging.

Installing python-simpy with apt-get. Apt-get proposes to the user a solution with 119 upgraded packages, and 203 newly installed, which might be acceptable for a user enjoying a lot of disk space. Unfortunately, the solution proposed by apt-get also requires to remove 414 installed packages, which is clearly unacceptable.

```
apt-get --without-recommends install python-simpy
The following extra packages will be installed:
  binutils ca-certificates coreutils cpp cpp-4.3
  ...
  tk8.4 totem-common whois xinit xkb-data zlib1g
The following packages will be REMOVED:
  abiword-common abiword-gnome amor apt apt-utils
  ...
  xserver-common xserver-xfree86 yelp zenity
The following NEW packages will be installed:
  ca-certificates cpp-4.3 dbus
  ...
  system-tools-backends totem-common whois
The following packages will be upgraded:
  binutils coreutils cpp debconf dia-libs
  ...
  ocaml-interp perl perl-base perl-modules
119 upgraded, 203 newly installed, 414 to remove
 and 338 not upgraded.
```

Installing python-simpy with aptitude. Aptitude's solution is more conservative with "only" 25 packages upgraded, 16 newly installed and 18 to remove.

```
aptitude --without-recommends install python-simpy
The following NEW packages will be installed:
  debian-archive-keyring{a} [2010.08.28~lenny1]
  ... python-simpy [1.8-1] ...
  python2.5-minimal{a} [2.5.2-15+lenny1]
The following packages will be REMOVED:
  aptitude{a} [0.2.15.9-2]  dia-common{a}
  ...
  synaptic{a} [0.55+cvs20050503-4]  tasksel{a}
The following packages will be upgraded:
  apt [0.5.28.6 -> 0.7.20.2+lenny2]
  ...
  zlib1g-dev [1:1.2.3.3.dfsg-12]
25 packages upgraded, 16 newly installed,
 18 to remove and 784 not upgraded.
```

Installing python-simpy with cupt or smart. The last two package managers in our list do not manage to find a solution in an acceptable time: `cupt` fails immediately, while `smart` enters into a long computation phase that we aborted after 10 minutes.

```
smart install python-simpy
Loading cache...
Updating cache...   ######################## [100%]

Computing transaction...
^C
```

2.2. Using MPM

As we will show in Section 3.2, the Mancoosi Package Manager (MPM) allows the user to specify high-level user-defined optimization criteria, which are used to choose a solution well adapted to the user needs, and which is computed by resorting to an efficient external solver.

On our example, calling MPM with a very conservative policy, namely to minimize the number of removed and changed packages, leads to a very simple solution which consists of installing an old version of the package (which might be sufficient for the user's needs).

```
mpm -c "-removed,-changed" install python-simpy
Summary of proposed changes:
 new: 2
 removed: 0
 replaced: 0
 upgraded: 0
 downgraded: 0
 unsatisfied recommends:: 97
 changed: 2
 uptodate: 128
 notuptodate: 824

New packages:
  python-simpy (1.5.1-2) python2.3-simpy (1.5.1-2)
```

The user may require a different version of the package by specifying it on the command line, as `python-simpy=1.8.1`. This allows to obtain a solution from MPM that can be directly compared to the ones found in the previous experiments run with the legacy package managers. In this case MPM provides a solution that is slightly better than the one found by `aptitude` with 15 new, 17 removed and 21 upgraded packages.

```
mpm -c "-removed,-changed" install python-simpy=1.8.1
Summary of proposed changes:
 new: 15
 removed: 17
```

```
 replaced: 1
 upgraded: 21
 downgraded: 0
 unsatisfied recommends:: 92
 changed: 54
 uptodate: 146
 notuptodate: 801
New packages:
 python-minimal (2.4.4-2) python-simpy (1.8-1)
 ...
 readline-common (5.2-3.1)
Removed packages: aptitude (0.2.15.9-2)
 dia-gnome (0.94.0-7sarge3) gnome (64)
 ...
 python2.3-numeric (23.8-1) synaptic
Replaced packages: linux-kernel-headers
Upgraded packages: apt [0.5.28.6] (0.7.20.2+lenny2)
 (0.7.20.2+lenny2) binutils [2.15-6]
 ...
 libxslt1.1 [1.1.12-8] (1.1.19-3)
```

In this particular example, the total running time of MPM is 10 seconds, 2 of which are spent by the solver to find a solution and 8 to write the CUDF file and read the solution. For comparison, `apt-get` took 3 seconds total running time, and `aptitude` 8 seconds. In order to compete with legacy solvers we will have to speed up the CUDF file handling between the external solver and the package manager component. However, since the solver component takes only a minor fraction of the running time, we are confident that these details can be easily addressed in a future release of MPM.

As we will see in the rest of the paper, this is not an isolated case: when confronted with installation problems mixing different repositories, MPM significantly outperforms all the legacy package managers.

3. Modular package management

Among all functionalities of a package manager, dependency solving is the most difficult, recurrent, and apparently underestimated one. Re-developing from scratch dependency solvers as soon as dependencies and conflicts are introduced in yet another component model seems to have not served well FOSS users thus far. We argue that an alternative, more modular, approach is possible by treating *dependency solving as a separate concern* from other component management issues. The goal is to decouple the evolution of dependency solving from that of specific package managers *and* component models.

To attain this goal we have designed a modular architecture (see Figure 1) which uses two well defined interfaces to interconnect the components: the first is a common format for describing package upgrade problems, CUDF that is both distribution-agnostic and neutral with respect to the solving technology.

The second is a simple, yet powerful optimization language for specifying common user requirements and selecting an optimal solution.

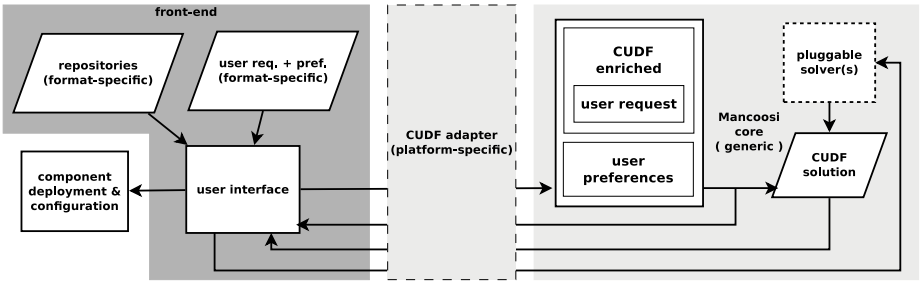

Figure 1: modular package manager architecture

This architecture brings back separation of concerns to package management design:

Conversion using a pivot format. CUDF adapters are developed and maintained once for each component model by the developers maintaining the package metadata format, or by CUDF experts working with them;

Reuse of dependency solvers. Solvers are created and maintained once by solver experts, who will see their technology gain many new fields of application by just supporting one generic I/O format—CUDF—which comes with a rigorous semantics, relieving the pain of interpreting the meaning of platform-specific component metadata.

Using this framework, the developer of a package manager for a specific component distribution only needs to be concerned with the front-end, the other parts being reusable components built and maintained only once, and evolving independently from the package manager.

In terms of architectural constraints, we notice that our proposal makes no assumptions on the localization of the different components: a distributed implementation might very well run a solver farm on a remote cluster, even if this would require to address reliability and fault tolerance issues more deeply than a centralized implementation.

In the next section we will present MPM, our own centralized implementation of the architecture of Figure 1, but first of all, we give here a detailed presentation of the CUDF format and of the user preferences language, as well as an overview of the existing solvers that already support CUDF.

3.1. A unified description of upgrades, using CUDF

Treating dependency solving as a separate concern in component upgrade planning requires a language able to capture all relevant aspects of upgrade problem instances. In this section we present a DSL called CUDF (for Common Upgradeability Description Format), whose documents describe instances of the

component upgrade problem. The design of CUDF has been guided by a few general principles:

Platform independence CUDF is a *common* format to describe upgrade scenarios coming from diverse environments. As a consequence, CUDF makes no assumptions on a specific component model, version schema, dependency formalism, or package manager. In fact, translators from different package formats have been implemented (see Subsection 3.1.3).

Solver independence In contrast to encodings of inter-component relations which are targeted at specific solver techniques (see Section 6), CUDF stays close to the original problem, in order to preserve its structure and avoid bias towards specific solver. The existence of solvers using CUDF as input format and that are based on very different solver technologies (Section 3.3) shows that this goal has been achieved.

Readability CUDF is a compact plain text format which makes it easy for humans to read upgrade scenario, and eases interoperability with package managers. As evidence of the benefits of this choice, CUDF is routinely used by the Eclipse P2 team to reason about upgrade scenarios, instead of the native XML encoding that comes with Eclipse[2].

Extensibility Only core component properties that are shared by mainstream package formats and that are essential to the meaning of upgrade scenarios are predefined in CUDF. Other auxiliary properties can be declared and used in CUDF documents, to allow the preservation of relevant information that can then be used in optimization criteria, e.g. component size, number of bugs, etc. As an example of additional properties demonstrating the extensibility of the format, Figure 2 shows both declaration and use of custom properties *bugs* and *suite*.

Formal semantics CUDF comes with a rigorous semantics that allows package manager and solver developers to agree on the meaning of upgrade scenarios. For example, the fact that self-conflicts are ignored is not a tacit convention implemented by some obscure line of code, but a property of the formal semantics. The formal semantics will be summarized below in Section 3.1.2.

3.1.1. Language overview

An upgrade scenario is represented by a *CUDF document*. It consists of a sequence of *stanzas*, each of which is a collection of key-value pairs called *properties*. Properties are typed within a simple *type system* containing basic data types (integers, booleans, strings) and more complex, component-specific

[2]See http://wiki.eclipse.org/Equinox/p2/Meetings/20091221, retrieved December 2011

data types such as boolean formulae over atomic package constraints used to represent inter-component relationships.

Each CUDF document is made up of three logical sections: a *preamble*, a component *universe*, and a *request*. The universe contains one *component stanza* for each component known to the package manager. Both installed and non-installed (but available) components are represented uniformly in the same document, in contrast to current platforms which often spread this information over different locations using different formats.

The properties used in package stanzas may be core properties of the CUDF format, or extra properties that have been declared in the preamble of the CUDF document. Properties may have a default value, and in this case are allowed to be absent from a stanza. The core properties of CUDF are:

`package` with values of type `string` denoting the name of a package (independent from a particular version).

`version` with non-negative integer values denoting the version of a package.

`depends` with values that are conjunctions of disjunctions of atomic package constraints. Dependencies express positive context requirements.

`conflicts` with values that are lists of atomic package constraints. Conflicts express negative context requirements.

`provides` with a list of equational package constraints. A provided package may be used to satisfy a dependency, and has to be taken into account for conflicts.

`installed` with boolean values indicating whether the package is currently installed or not.

`keep` with values indicating whether the package is allowed to be removed or updated.

An atomic package constraint consists of a package name and a unary predicate on version numbers, like the empty predicate (which is true for all versions), $= 17$, $\neq 17$, > 42 or ≤ 17. Equational package constraints are only allowed to use the empty predicate, or an equality on the version number like $= 25$.

Figure 2 shows a sample CUDF document. The component universe contains several component stanzas, where both core and extra properties are used. Extra properties must be declared in the *preamble*, which starts the document. Extra properties account for extensibility of the format and enable type checking of CUDF documents. A *request stanza* encodes the user request and concludes the document. In its general form, the request stanza details the components the user wants to `install`, `remove`, or `upgrade` (using the homonymous properties), possibly specifying version requirements. The value of such a request is a list of atomic package constraints.

Appendix A defines the full syntax of CUDF in form of an EBNF grammar.

```
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
```

Figure 2: Sample CUDF document

3.1.2. Overview of CUDF formal semantics

In order to simplify the description of the semantics we ignore here the `keep` property, and take into account only installation and removal requests (but no upgrade requests). A solution to a CUDF document U is a subset of the universe of U, consisting of the packages that are installed on a system as a result of the upgrade operation. For any subset S of the universe of U, let $dom(S)$ be the set of pairs of package name and version of its packages (in a valid CUDF document, package name and version uniquely identify a package). Furthermore, let $pro(S)$ be the set of pairs (name, version) that are provided by packages in S. The version of a provided package is either the one explicitly mentioned, or any version in case the version predicate is empty (the set $pro(S)$ may, as a consequence, be infinite).

One defines easily what it means for a set of pairs (name,version) to satisfy an atomic package constraint. Then, a subset S of the universe of U is

- **abundant** if every disjunction in the dependency of every package in S contains a package constraint that is satisfied by $dom(S) \cup pro(S)$;
- **peaceful** if no atomic package constraint in the conflicts of any package $p \in S$ is satisfied by $dom(S - \{p\}) \cup pro(S - \{p\})$,

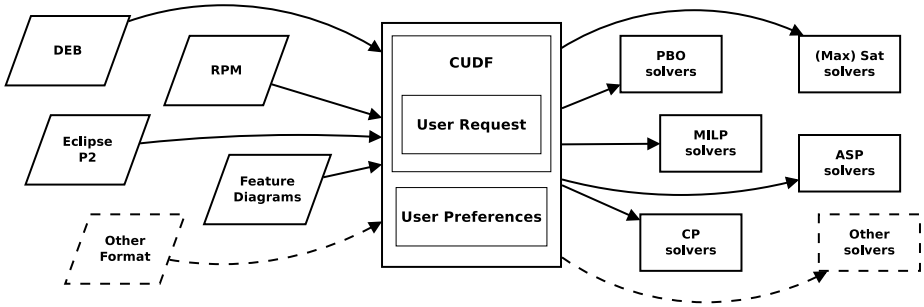

Figure 3: Sharing upgrade problems and solvers among communities

A subset S of the universe of U satisfies a request

- `install` ϕ if every atomic constraint in ϕ is satisfied by $dom(S) \cup pro(S)$;
- `remove` ϕ if no element of ϕ is satisfied by $dom(S) \cup pro(S)$.

S is a solution if it is abundant, healthy, and satisfies the request. Note that a solution is allowed to contain several packages with the same name (which then have to differ in version). Also, there is a subtlety in the definition of health, in that a package (and the packages provided by it) are not taken into account when evaluating its own conflicts.

The full formal semantics of CUDF is given in Appendix B.

3.1.3. Expressiveness

As CUDF lays at the "interface" between package managers and dependency solvers, its expressiveness should be validated looking from both angles. From the point of view of package managers, we have shown that upgrade scenarios coming from several major component models can be encoded in CUDF; adapters are already available for: Debian (see Appendix C) and RPM (see Appendix D) packages, Eclipse [6]—with an extension for full OSGi bundles in the working— and common feature diagram formalisms used in software product lines [7]. All encodings are *linear* in the number of components to encode, even in the presence of XOR dependencies.[3]

From the converse angle, that of dependency solvers, we observe that entrants in the MISC competition [3] have used very different solver technologies: boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), and graph constraints. They have all been able to handle upgrade problems encoded as CUDF documents, providing convincing evidence that CUDF is adequate for a large spectrum of solving techniques.

Hence, at the time of writing, CUDF is already a unique pivot format that allows on the one hand to share solvers among different package managers, and

[3] while usual SAT encodings blow up quadratically in the number of XOR dependencies.

on the other hand to share a corpus of challenging upgrade problems among solver communities, as shown in Figure 3. The number of supported solver technologies and component frameworks has grown steadily over the past years and it is likely to keep growing in the future.

3.1.4. Implementations

CUDF has been subject to an ad-hoc standardization process, resulting in a specification [24]. `libcudf` is the "reference" implementation of the specification; it consists of a parsing and pretty-printing library for CUDF, as well as an implementation of CUDF semantics. The latter consists in:

1. given a CUDF document, `libcudf` can verify whether installed components are consistent, i.e. whether they satisfy abundance and peace;
2. given a CUDF document and an encoding of a potential solution, `libcudf` can verify whether the solution is valid, i.e. abundance, peace, and request satisfaction.

`libcudf` comes with the `cudf-check` command line tool which provides the above two features out of the box. `libcudf` is Free Software and can be used both from the OCaml and C programming languages; it is available at `http://www.mancoosi.org/software/`.

The authors are aware of other CUDF implementations. Some have been developed in the context of the Mancoosi project to capture FOSS distribution upgrade scenario descriptions into CUDF, in order to build a cross-distribution corpus of upgrade problem instances [4]. Using the tools we have verified that the average size of an upgrade scenario encoded in CUDF is linear with the size of the original package manager information and usually smaller, since metadata not relevant for describing the upgrade problem can be dropped. For instance, on a large Debian installation, using both testing and unstable suites (totaling $\approx$ 45.000 packages), APT information on disk amounts to 14 Mb while the corresponding CUDF document is only 9 Mb.

An independent CUDF implementation is also available in CUPT [17], a new APT-compatible package manager for Debian. In CUPT, CUDF is used as an interface format to pipe upgrade scenarios to external solvers, so that upgrade planning can be decoupled from other package manager activities. While no stable software has been released yet, work is ongoing to implement CUDF in APT and APT2 in order to decouple dependency solving from the package managers.

3.2. User preferences as multicriteria optimization

The DSL presented in the previous section addresses the need of grasping those aspects of an upgrade scenario that are related to the *correctness* of a given solution (i.e. "does the solution satisfy the user request as well as the expectations of all installed components?"). *Quality* aspects of solutions (i.e. "is the proposed solution to my liking?") are much less known, not to mention agreed upon, and hence they do not yet constitute suitable material for DSL

standardization. Nonetheless, to improve the state of the art in upgrade planning we do need at the very least a rigorous framework to reason about solution quality. In this section we propose one such formalism.

As we have seen in Section 2 it is necessary to allow users to express their preferences about the desired solution. The state-of-the-art approach is to present one particular solution—found according to some built-in strategies generally unknown to the user—and then allow the user to interactively fiddle with the solution. This, however, may lead to an unacceptable burden on the user. The solver may, for instance, propose a solution that upgrades as many packages as possible to the latest available version, while the user might actually prefer a solution that changes as few packages as possible. In this case, an interactive correction of the solution will require many manual steps.

An alternative approach is to let the user specify high-level criteria that capture what she considers important in a solution: she may be concerned about the packages that are *changed*, the packages that are *not up to date*, the packages that get *removed*, or even "the number of installed security fixes", or "the overall installed size". On top of CUDF semantics, we can build an extensible dictionary of well-defined criteria like the above and then let the user inform the solvers that the required solution should maximize, or minimize, a given criterion.

It is quite natural for the user to combine several of these criteria: to compare two solutions s and s' whose criteria have values $(c_1, \ldots, c_n)$ and $(c'_1, \ldots, c'_n)$, the user will prefer s over s' if all criteria of s are better or equal than s' (i.e. s is Pareto-better then s'). Unfortunately, when one has more than one criterion, there may be many incomparable Pareto-optimal solutions; this is the core problem of *multicriteria optimization* which has been extensively studied in the optimization research community [20]. Many different approaches have been proposed to aggregate multiple criteria, the most common being:

Lexicographic The criteria are ordered by importance, and compared lexicographically: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff there exists an i s.t. for all $j < i$ $c_j = c'_j$, and $c_i > c'_i$; for example, a security upgrade may be considered more important than any other criterion, and put first in the order.

Weighted sum The criteria are aggregated into a single measure using user-specified *weights* k_i: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff $\sum_{1 \leq i \leq n} k_i c_i > \sum_{1 \leq i \leq n} k_i c'_i$; this may be useful when trying to balance different criteria for which no clear order is established.

More sophisticated approaches exists, like *leximin* and *leximax* [10], and an extensive literature is devoted to them. According to the use case, the best choice of an aggregation function may vary widely. Our own proposal for a high level user preferences formalism is simple yet expressive:

1. define a dictionary of useful criteria c_i;
2. define a dictionary of aggregation functions *lex*, *weightedsum*, *leximin*, etc.

Table 1: Optimization criteria

$removed(I,S)$	=	$\{name \mid V(I,name) \neq \emptyset$ and $V(S,name) = \emptyset\}$
$new(I,S)$	=	$\{name \mid V(I,name) = \emptyset$ and $V(S,name) \neq \emptyset\}$
$changed(I,S)$	=	$\{name \mid V(I,name) \neq V(S,name)\}$
$notuptodate(I,S)$	=	$\{name \mid V(S,name) \neq \emptyset$ and does not contain the most recent available version of $name\}$
$unsatrec(I,S)$	=	$\{(name,v,c) \mid v$ is an element of $V(S,name)$ and $(name,v)$ recommends $...,c,...$ and c is not satisfied by $S\}$
$sum(name)(I,S)$	=	$\sum_{p \in S} p.name$

3. write the user preference as an expression $op(k_1c_1,\ldots,k_nc_n)$ where k_i can be one of $\{+,-\}$ to indicate maximization or minimization of the criterion (for aggregation functions like *lex*, *leximin* and *leximax*), or an integer (for aggregation functions like *weightedsum*).

Formally we define the criteria as in Table 1, where I is the initial installation and S is a proposed new installation. We write $V(X,name)$ the set of versions in which *name* (the name of a component) is installed in X, where X may be I or S. That set may be empty (*name* is not installed), contain one element (*name* is installed in exactly that version), or even contain multiple elements in case a component is installed in multiple versions.[4] Using this formalism, it is quite easy to define a *paranoid* preference as

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

The solution scoring best under this criterion will be the one with the minimum number of removed functionalities, and then with the minimum number of changes. A *trendy* preference is also easy to write

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

The last optimization criterion, $sum(name)$ is the sum over all installed packages of the value of their field name, where name is a property of type int or of one of its subtypes $(nat, posint)$. This property can be one of the core properties defined in the CUDF specification, or a property that is declared in the preamble of the CUDF document.

Currently, each criterion and aggregation function must be specifically encoded for a given solver technology, but work on a generic system which will be able to produce these encodings automatically is ongoing [25].

3.3. Dependency solvers

Any dependency solver that uses CUDF as input and output format, and that accepts the optimization criterion language described in Section 3.2 can

[4]The CUDF component allows to install multiple versions of a package, so-called *flat* models that allow for only one version per package to be installed can be encoded [4].

be used in conjunction with MPM. CUDF is precisely the input format of the *Mancoosi International Solver Competition (MISC)* [3], and the optimization criterion language has been introduced into MISC since November 2010. This competition has been launched in 2010 with the goal to bring together modern problem solving techniques with the challenging optimization problems that arise from component installation management.

Six different solvers participated in the 2010 and 2011 editions of the MISC competition, and use CUDF as input and output format:

1. `apt-pbo` [26], using a PBO solver
2. `aspcud` [11], based on Answer Set Programming using Potassco, the Potsdam Answer Set Solving Collection
3. `inesc` [5], using the p2cudf parser (from Eclipse) and the MaxSAT solver MSUnCore
4. `p2cudf` [15, 16], a family of solvers on top of the Eclipse Provisioning Platform p2, based on the SAT4J library
5. `ucl`, based on graph constraints
6. `unsa` [19], based on ILOG's CPLEX

Since November 2010, four of these solvers already accept the user preference language, and are hence eligible as solver plugin to MPM. The current prototype implementation of MPM employs `aspcud`.

4. The Mancoosi package manager

The Mancoosi package manager (MPM) is a proof-of-concept implementation which integrates solver technology and optimization criteria to solve real world installation problems. The back-end of MPM leverages the infrastructure of the `apt` package manager both to parse command line arguments and to perform package installation, but is modular with respect to the dependency solver component.

To facilitate the acceptance of MPM we decided to maintain a strict compatibility with existing tools. In particular, MPM has been developed to be used as a drop-in replacement for `apt-get`/`aptitude` and to provide an easy-to-use alternative to solve complicated problems for which the apt solvers are unable to find a solution.

4.1. Using MPM

Currently, four different utility functions are defined that can be used in the definition of the optimization criterion. These are defined in terms of the effect that realizing a proposed solution would have:

removed is the number of packages that would be removed by a solution;

new is the number of packages newly installed by a solution;

changed is the number of packages that would be changed by a solution (that is, removed, newly installed, upgraded or downgraded)

notuptodate is the number installed packages that, after installation of the solution, would be installed in a version that is older than the latest known version;

The current implementation of MPM allows a user to

- specify a solver plugin with the option `-s <solver>`, and
- specify an optimization criterion with the option `-c <option>`, where `<option>` is either one of the predefined criteria *paranoid* or *trendy*, as defined in Section 3.2 or an arbitrary optimization criteria specified by giving a lexicographic combination of the available utility functions.
- Besides, any option argument to the `-o` option is passed as an option to `apt`.

The actual request may be any of `install`, `remove`, or `upgrade`, together with a list of package names (possibly with a constraint on the respective version of the package).

A user may very well use different optimization criteria for upgrade request than for install/remove requests (upgrade means possibly replacing a package by a newer version). This is due to the fact that certain criteria that are useful with one type of requests may just not make sense when combined with another type of requests. For instance, the *paranoid* criterion defined in Section 3.2 may be chosen by a careful sysadmin when installing or removing packages, but it is certainly not useful for upgrade request: due to the CUDF semantics, not changing at all the installation status of packages would be a correct solution to any upgrade request of packages that are already installed, and it would even be the optimal solution. This is certainly not what the user had in mind. When issuing an upgrade request one most likely will chose an optimization criterion that puts high priority on minimizing the number of not-up-to-date packages.

4.2. Implementation

The implementation of MPM makes use of some of the components and tools that have been developed by the Mancoosi project, and that are centered around the CUDF format. Since MPM is targeted to Debian we use `debtodudf` [4] to translate Debian package metadata to CUDF; translators to CUDF are also available for the metadata format of `rpm` [4] and Eclipse [6]. MPM communicates with the CUDF solvers via `apt-cudf` which translates a context-specific exchange format (EDSP) to CUDF and returns the solution from the solver to the MPM in the same format. Acting as a pipe, `apt-cudf` runs during the entire solving phase to reduce the translation overhead. The EDSP format is a simple text based encoding of the Packages universe and the user request.

MPM is written in python and uses the python bindings to libapt to access the `apt` API. For this reason MPM maintains full backward compatibility with apt, reusing the same configuration files and settings.

MPM supports all command-line options of `apt` (see the `apt-get` (1) man page). In addition, the user may specify a solver and an optimization criterion to be used to satisfy the current request.

```
APT::Solver::name "aspcud-paranoid-1.3";
APT::Solver::criteria "-removed,-changed";
```

Since MPM aims to replace only the dependency solver part of apt, everything else will still be handled by the underlying library. In particular, once a solution is found by the dependency solver, apt will be in charge of finding an "installation plan" for the proposed solution. This installation plan defines the order in which packages will be installed on the system, and specifically the invocation order of dpkg, the low level debian package installer.

As a consequence of the separation of concerns allowed by our architecture, the size of the code of the MPM prototype itself is rather small, with less than 500 lines of Python.

5. Experimental validation

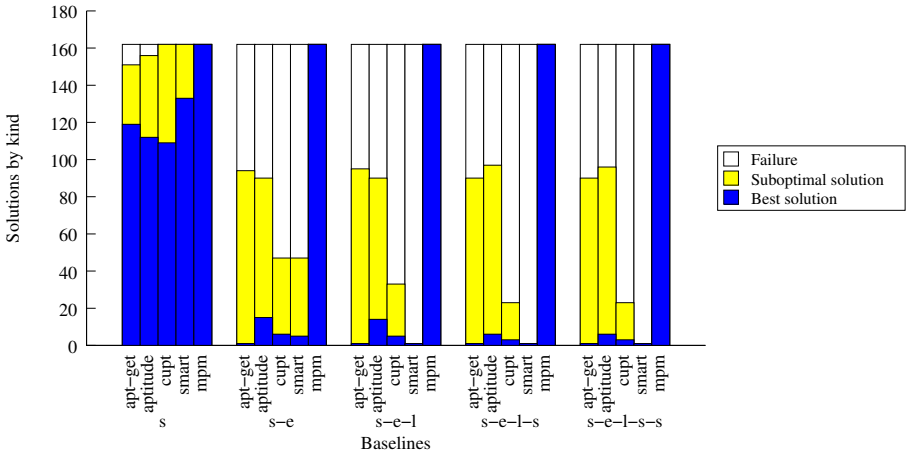

Figure 4: Comparison of mainstream meta-installers with MPM on increasingly large package universes. Baselines are formed taking increasingly large sets of Debian releases, indexed by the first letter of their name: sarge, etch, lenny, squeeze, sid.

We compared MPM to the latest version available in Debian of four different state of the art packages managers. Our goal was to assess the improvements in the quality of the solution that are attainable using our modular architecture by reusing solvers which participated in the latest Mancoosi International Solver Competition (MISC 2011) [1, 3].

For this particular experiment, MPM has been configured to use the `aspcud` solver, one of the winners of the MISC 2011 competition: based on Answer Set Programming, this solver is available under an Open Source license and is readily available in the Debian distribution.

We did not take execution time into account for two reasons: on the one hand side, MPM is a prototype which is not optimized for reading, writing and caching package repositories; on the other hand, as the results show, on difficult

problems MPM is the only package manager able to find a solution, so there is nothing to compare with.

Experimental environment. Since performance was not a goal of this study, our simulation environment was established in a clean Debian chroot on a commodity x86 machine. All relevant tools and raw data are available online at http://data.mancoosi.org/papers/ist2012/.

Method. We performed five groups of tests, using the same installation / removal requests with a combination of different Debian releases - or baselines - (`sarge`, `etch`, `lenny`, `squeeze` and `sid`). The initial *status* for these experiments was selected as a set of installed packages on a Debian server running `sarge`.

The package managers selected for these tests were `apt-get` 0.9.7, `aptitude` 0.6.7, `smart` 1.4, and `cupt` 2.5.6. The first two package managers are the most representative ones as they are the standard tools in Debian; the last two were selected because they were available at the time of writing in the Debian distribution and at the same time capable of working with Debian metadata. All package managers were used with their default options.

All these legacy solvers have some kind of optimization criterion hardwired in their implementation. Hence, it is not obvious how to compare them in a fair manner. Since the algorithms of these tools all work locally (they try to satisfy the user request by only looking at the dependencies directly related to the request) we decided to compare their solutions with respect to the *paranoid* optimization function described in Section 3.2 which privileges conservative solutions.

In order to compare solutions from different solvers we reused tools that were originally developed for the MISC solver competition [1], by converting the package-manager's output into a CUDF solution and then comparing them using the *paranoid* optimization criterion.

Experimental Data. A problem instance for our test is composed of three parts: an installation status (a set of packages which are already installed on the machine that the users wants to modify), a universe of packages available for installation, and a user request (which packages to install or remove).

In all our problem sets, the installation status is the same, and is the image of the installation of a standard server running Debian sarge. We used five different universes of packages, obtained by progressively adding together, in chronological order, all the packages available in the last five Debian baselines: `sarge`, `etch`, `lenny`, `squeeze` and `sid`. The smallest universe, with only `sarge`, contains 15.000 packages, while the largest universe, with the union of all baselines, adds up to 60.000 packages.

On each of these five universes, we have run 162 user requests, one half of them requiring the installation of 5 packages and the other half requiring the removal of 5 packages. Test problems are generated by choosing 5 packages at

random from the initial installation status[5]. To provide a realistic scenario, we ensure that all the problems have a solution.

Package managers were tested with different timeouts, ranging from 60 to 300 seconds for each request. Since there are no noticeable differences, in the following we only show the results for the run executed with a 300 seconds timeout.

Assessment. In Figure 4 we show the aggregate results of our tests where solutions are ranked according to the "paranoid" criterion and then divided into three categories, *best solution*, *sub-optimal* and *failure*. The first two categories contain respectively the best solutions found (including ex-aequos) and not optimal, but still correct solutions. Note that *best solution* here means the best solution among all solutions delivered by all participating solvers, and not necessarily the theoretically best possible solution, which we do not know.

The failure category aggregates all results that were either not a solution or were the result of a timeout or a crash of the package manager.

When looking for a solution in a universe containing only the `sarge` baseline, all solvers find a solution in most cases, and this solution is optimal in roughly 75% of the cases, with the *smart* approaching the performance of MPM.

The situation changes radically as soon as more than one baseline is considered: all legacy package manager fail to find a solution in almost 50% of the cases. This is consistent with the folklore experience of FOSS distribution users who know that maintaining an *old* machine using these tools becomes more difficult over time.

On the other side, MPM is remarkably stable, and consistently finds the best solution, no matter what the composition of the universe is. This is not a surprise, as MPM employs a state-of-the-art external solver. This solver is designed to find a *global* solution that is optimal with respect to the paranoid criterion, unlike the legacy package managers which apply ad-hoc local search algorithms.

These results also justify the fact of not comparing execution times: when the maintenance problem become difficult, MPM is the only viable tool, even in its prototype, non-optimized form.

6. Related work

6.1. State of the art package managers

The world of FOSS distributions has grown very complex over time: the dedicated page[6] on Linux Weekly News lists more than 600 of them. Despite this large variety, most distributions use one of two mainstream package formats, RPM and DEB, originally designed for the RedHat and Debian distributions respectively, but now largely adopted by most of the others.

[5] The test generation suite is available as part of the dose3 software library

[6] `http://lwn.net/Distributions/`, retrieved December 2011

Each of these two different package formats comes with standard tools for the low level operations on packages, namely `rpm` and `dpkg`. One finds a greater variety of *package managers*, that is tools for higher level operations like fetching packages, validating their signatures, solving dependencies, and planning upgrades. The most used package managers in the RPM universe are: `urpmi`, `yum`, `yast`, and `apt-pbo`. `Urpmi` and `yum` are front-ends to `rpm` and have a monolithic architecture and a custom dependency solver. `Yast`, on the other hand, uses an external satisfiability solver (`libzypp`), but preferences are hard coded in the problem encoding [21, 22]. `apt-pbo` [26] is a fork of `apt-get` (or `apt` for short) and uses a pseudo-Boolean solver as dependency solving engine. It is possible to specify custom optimization functions to reflect user preferences.

The Debian ecosystem of package managers is less fragmented with `apt-get` and `aptitude` playing the main role, and many other alternative package managers striving to gain acceptance in the community. Notably, the `cupt` package manager [17] in its latest revision implements an experimental CUDF back-end. `apt-get` and `aptitude` are both based on libapt, `apt-get`'s low-level library used to handle configuration files, package retrieval, and installation planning. Both `apt-get` and `aptitude` contain an ad-hoc dependency solver. `apt-get` has experimental support to use an external SAT solver, however it is still not possible to specify user criteria directly.[7]

One exception to this classification is the cross distribution package manager `smart`. It has a modular architecture with respect to the package installer backend and the metadata format. The notion of a best solution is specified by a policy which assigns weights to different solutions, and thus allows `smart` to choose the most suitable one. By trying to obtain an optimal solution, the `smart` dependency solving algorithm explores a potentially huge solution space, using some heuristics to avoid getting lost.

The problem of finding an optimal installation candidate, with respect to some criteria, is computationally hard and is treated differently by different tools. Some rely on special heuristics, like `apt-get` 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. These tools may fail 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 requiring unacceptably high computation time in some cases.

Table 2 summarizes the characteristics of the tools we compared in our work; the fact that `apt-get`, `cupt` and `aptitude` are not complete is shown in the experimental results of Section 5, where several problem instances that admit a solution are not solved by these tools. The completeness of `smart` is an open

[7] as discussed among authors and `apt-get` developers starting at `http://lists.debian.org/deity/2010/05/msg00088.html` and implemented starting from `http://packages.qa.debian.org/a/apt/news/20110629T214718Z.html` (URLs retrieved December 2011)

Tool	Solver	Optimization	Complete
`apt-get`	internal	hard-coded	no
`aptitude`	internal	hard-coded	no
`cupt`	internal	hard-coded	no
`smart`	internal	programmable	maybe

Table 2: Summary comparison of the different package managers

question: in our experiments we did not find a counterexample, but the tool times out very often.

6.2. Formal approaches

Several attempts at using formal techniques to attack dependency solving problems have been made in the past; in some cases they resulted in the development of package manager prototypes.

A first formal encoding of the upgrade problem in SAT, together with a proof of the NP-Completeness of the problem, has been initially proposed in the context of the EDOS project [9, 18]. Following that initial encoding, the use of SAT technology in package managers has seen a raise in popularity. The OPIUM system [27] used in 2006 a SAT solver with an optimization in line with our paranoid criterion, SUSE libzypp [21] incorporated a SAT solver in 2007, the Eclipse P2 system includes the Sat4J solver since 2007 [15]. This trend seems to continue steadily: a very recent entrant is apt-pbo, introduced in the Caixa Mágica GNU/Linux distribution just a few months ago [26]. In this paper we have not benchmarked against some of them for applicability reasons, in particular: apt-pbo as of now does not support removal requests (a quite severe limitation) while, in contrast to our benchmark data, libzypp is targeted at RPM packages.

Several alternative encodings of the upgrade problem have been proposed: SAT [18, 27, 15], Pseudo Boolean Optimization [26], Partial Weighted Max SAT [5], Mixed Integer Linear Programming [19], as well as some others championed by entrants in the 2010 and 2011 editions of the MISC competition (see Section 3.3).

Jenson [13] proposes a component model without explicit (or implicit) component conflicts and does not handle component removal in neither requests nor solutions. As a consequence, such a degenerate upgrade problem is way simpler than the problem considered in this paper and can be solved in polynomial time, even though the number of solutions may be huge. Dependency solving as SAT with optimization has been reviewed in [15] where it was also observed that much of the complexity stems from multiple versions of components and the constraints they entail.

To the best of authors knowledge, the key idea of making dependency solvers a modular component of package managers only appeared in [2], which is an earlier version of this work. This journal article provides more extensive discussion of the relevance of the approach and related works, and adds a self-contained

presentation of the CUDF language, all the details about the conversion process into CUDF from the metadata found in various FOSS distributions, and more up-to-date experimental results.

7. Conclusions

We have presented in this work a modular package manager architecture that allows to rely on external state-of-the-art solvers for dependency handling, thanks to the formally defined CUDF format coming from the MISC solver competition. Our architecture also provides the user with a flexible, high-level preference language that allows to tailor the solution to one's needs.

We have built a proof-of-concept package manager, called MPM, for Debian-based FOSS distribution. MPM is based on the architecture proposed in this paper, and makes use of the modular components already available: solvers coming from the MISC competition, converters between the Debian package format and CUDF, and low level package management libraries for Debian coming from the `apt` library.

Despite the fact that MPM is only a proof of concept, it significantly outperforms all legacy package managers, as soon as installation problems with more than a single release are to be solved.

These results fully validate our approach, and MPM may already be used as a drop-in replacement of legacy package managers for complex installation or upgrade tasks on which legacy tools fail.

Acknowledgments

The authors are grateful to the members of the Mancoosi project for many stimulating discussions on the CUDF format, optimization criteria, and the practical use cases. A special acknowledgment goes to the optimization and solving community who took part in the MISC competition, adapting their solver to the CUDF format: this allows the direct reuse of their solvers in the MPM package manager, paving the way to a general reuse in other package managers based on the modular architecture proposed here. The comparison of packages with software component models given in Section 1 was inspired by a discussion with Kung-Kiu Lau. We thank the anonymous referees for their constructive help in improving the presentation.

Data availability

All tools and raw data used for the validation of the results shown in this paper are available online at http://data.mancoosi.org/papers/ist2012/. A synthetic presentation of the results can be found at http://www.mancoosi.org/measures/packagemanagers/2012.

References

[1] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., 2011a. MISC competition 2011. http://mancoosi.org/misc. Results announced on September, 2011.

[2] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., 2011b. MPM: a modular package manager, in: Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering, ACM, New York, NY, USA. pp. 179–188.

[3] Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., 2012. Dependency solving: A separate concern in component evolution management. Journal of Systems and Software October 2012.

[4] Abate, P., Guerreiro, A., Laurière, S., Treinen, R., Zacchiroli, S., 2010. Extension of an existing package manager to produce traces of ugradeability problems in CUDF format. Mancoosi deliv. D5.2. http://www.mancoosi.org/reports/d5.2.pdf.

[5] Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P., 2010. Solving Linux upgradeability problems using boolean optimization, in: LoCoCo, pp. 11–22.

[6] Bozman, C., 2010. Converting Eclipse metadata into CUDF. Technical report 5. Mancoosi project. http://www.mancoosi.org/reports/p2.pdf.

[7] Di Cosmo, R., Zacchiroli, S., 2010. Feature diagrams as package dependencies, in: SPLC: Software Product Lines Conference, pp. 476–480.

[8] EDOS Project Workpackage 2 Team, 2005. Report on Formal Management of Software Dependencies. EDOS Project Deliverable Work Package 2, Deliverable 1.

[9] EDOS Project Workpackage 2 Team, 2006. Report on Formal Management of Software Dependencies. EDOS Project Deliverable Work Package 2, Deliverable 2.

[10] Fishburn, P.C., 1974. Lexicographic orders, utilities and decision rules: A survey. Management Science 20, 1442–1471.

[11] Gebser, M., Kaminski, R., Schaub, T., 2011. aspcud: A linux package configuration tool based on answer set programming, in: Drescher, C., Lynce, I., Treinen, R. (Eds.), Proceedings Second Workshop on Logics for Component Configuration, Perugia, Italy, 12th September 2011, Open Publishing Association. pp. 12–25.

[12] Jackson, I., Schwarz, C., 2008. Debian policy manual. http://www.debian.org/doc/debian-policy/.

[13] Jenson, G., Dietrich, J., Guesgen, H., 2010. An empirical study of the component dependency resolution search space, in: CBSE, Springer. pp. 182–199.

[14] Lau, K.K., Wang, Z., 2007. Software component models. IEEE Trans. Software Eng. 33, 709–724.

[15] Le Berre, D., Parrain, A., 2008. On SAT technologies for dependency management and beyond, in: SPLC (2), pp. 197–200.

[16] Le Berre, D., Rapicault, P., 2009. Dependency management for the Eclipse ecosystem, in: IWOCE 2009, ACM. pp. 21–30.

[17] Lyubimkin, E.V., 2010. Cupt package manager. http://wiki.debian.org/Cupt.

[18] Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R., 2006. Managing the complexity of large free and open source package-based software distributions, in: ASE 2006, IEEE. pp. 199–208.

[19] Michel, C., Rueher, M., 2010. Handling software upgradeability problems with MILP solvers, in: LoCoCo 2010: Logics for Component Configuration, pp. 1–10.

[20] Steuer, R.E., 1986. Multiple Criteria Optimization: Theory, Computation and Application. Wiley.

[21] SUSE, 2008a. Libzypp library. http://en.opensuse.org/Portal:Libzypp.

[22] SUSE, 2008b. Yast package manager. http://en.opensuse.org/Portal:YaST.

[23] Treinen, R., Zacchiroli, S., 2008. Solving package dependencies: from EDOS to Mancoosi, in: DebConf 8: proceedings of the 9th conference of the Debian project.

[24] Treinen, R., Zacchiroli, S., 2009a. Common Upgradeability Description Format (CUDF) 2.0. Technical Report 3. The Mancoosi Project. http://www.mancoosi.org/reports/tr3.pdf.

[25] Treinen, R., Zacchiroli, S., 2009b. Expressing advanced user preferences in component installation, in: IWOCE 2009, ACM. pp. 31–40.

[26] Trezentos, P., Lynce, I., Oliveira, A., 2010. Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization, in: ASE'10, ACM. pp. 427–436.

[27] Tucker, C., Shuffelton, D., Jhala, R., Lerner, S., 2007. OPIUM: Optimal package install/uninstall manager, in: ICSE'07, IEEE. pp. 178–188.

Appendix A. CUDF syntax

Overall structure.

$$cudf ::= preamble?\ universe\ request$$

Flow elements.

$$\begin{aligned} ssep &::= (comment \mid \text{`\textbackslash n'})*\ \text{`\textbackslash n'}\ (comment \mid \text{`\textbackslash n'})* \\ comment &::= \text{`\#'}\ line \\ line &::= \texttt{[\^{}\textbackslash n]}*\ \text{`\textbackslash n'} \end{aligned}$$

Document parts.

$$\begin{aligned} preamble &::= \text{`preamble: '}\ line\ stanza\ ssep \\ universe &::= package* \\ package &::= \text{`package: '}\ pkgname\ stanza\ ssep \\ request &::= \text{`request: '}\ line\ stanza\ comment* \end{aligned}$$

Stanzas.

$$\begin{aligned} stanza &::= (property\ \text{`\textbackslash n'} \mid comment)* \\ property &::= propname\ \text{`: '}\ value \\ propname &::= ident \\ value &::= bool \mid enum \mid int \mid nat \mid posint \mid string \mid pkgname \mid ident \mid typedecl \\ &\quad \mid vpkg \mid veqpkg \mid vpkgformula \mid vpkglist \mid veqpkglist \end{aligned}$$

Values: CUDF types.

$$\begin{aligned} bool &::= \text{`true'} \mid \text{`false'} \\ int &::= (\text{`+'} \mid \text{`-'})?\ \texttt{[0-9]}+ \\ string &::= \texttt{[\^{}\textbackslash r\textbackslash n]}* \\ vpkg &::= pkgname\ (sp+\ vconstr)? \\ vpkgformula &::= andfla \mid \text{`true!'} \mid \text{`false!'} \\ vpkglist &::= \text{`'} \mid vpkg\ (sp* \text{`,'} sp* vpkg)* \\ enum &::= ident \\ pkgname &::= \texttt{[A-Za-z0-9+./@()\%-]}+ \\ ident &::= \texttt{[a-z][a-z0-9-]}* \\ nat &::= \text{`+'}\texttt{[0-9]}+ \\ posint &::= \text{`+'}\texttt{[0-9]}*\texttt{[1-9][0-9]}* \\ veqpkg &::= pkgname\ (sp+\ veqconstr)? \\ veqpkglist &::= \text{`'} \mid veqpkg\ (sp*\ \text{`,'}\ sp*\ veqpkg)* \\ typedecl &::= \text{`'} \mid typedecl1\ (sp*\ \text{`,'}\ sp*\ typedecl1)* \end{aligned}$$

Value: gory details.

```
   vconstr::=reop sp+ ver
 veqconstr::='=' sp+ ver
     relop::='=' | '!=' | '>=' | '>' | '<=' | '<'
        sp::=' ' | '\t'
       ver::=posint
    andfla::=orfla (sp* ',' sp* orfla)*
     orfla::=atomfla (sp* '|' sp* atomfla)*
   atomfla::=vpkg
 typedecl1::=ident sp* ':' sp* typeexpr(sp* = sp* '[' value* ']')?
  typeexpr::=typename | 'enum' sp* '[' ident (',' sp* ident)* ']'
 typename::='bool' | 'int' | 'nat' | 'posint' | 'string' | 'pkgname'
           | 'ident' | 'vpkg' | 'veqpkg' | 'vpkgformula' | 'vpkglist'
           | 'veqpkglist'
```

Appendix B. CUDF formal semantics

The semantics is defined in a style similar to [18], however, we now have to deal with an abstract semantics that is closer to "real" problem descriptions, and that contains artifacts like *features*. This induces some complications for the definition of the semantics. In [18] this and similar problems were avoided by a pre-processing step that expands many of the notions that we wish to keep in the CUDF format.

Appendix B.1. Abstract syntax and semantic domains

For any of the CUDF types t defined in Appendix A we denote by $\mathcal{V}(t)$ its semantic domain. For instance, $\mathcal{V}(\texttt{posint})$ is the set of positive natural numbers. If X is any set then we write $P(X)$ for its powerset. In addition, we give the following definitions:

Definition 1.

- CONSTRAINTS *is the set of version constraints, consisting of the value* $\top$ *and all pairs* $(relop, v)$ *where relop is one of* $=, \neq, <, >, \leq, \geq$ *and* $v \in \mathcal{V}(\texttt{posint})$.
- KEEPVALUES *is the set of the possible values of the* `keep` *property of package information items, that is:* $\{\texttt{version}, \texttt{package}, \texttt{feature}, \texttt{none}\}$.

The abstract syntax of a CUDF document is a pair consisting of a package description (as defined in Definition 2) and a request (see Definition 4).

Definition 2 (Package description)**.** *A* package description *is a partial function*

$$\mathcal{V}(\texttt{ident}) \times \mathcal{V}(\texttt{posint}) \quad \rightsquigarrow$$
$$\mathcal{V}(\texttt{bool}) \times \text{KEEPVALUES} \times \mathcal{V}(\texttt{vpkgformula}) \times \mathcal{V}(\texttt{vpkglist}) \times \mathcal{V}(\texttt{veqpkglist})$$

The set of all package descriptions is noted DESCR*. If* ϕ *is a package description then we write* $Dom(\phi)$ *for its domain. If* $\phi(p, n) = (i, k, d, c, p)$ *then we also write*

- $\phi(p,n).\texttt{installed} = i$
- $\phi(p,n).\texttt{keep} = k$
- $\phi(p,n).\texttt{depends} = d$
- $\phi(p,n).\texttt{conflicts} = c$
- $\phi(p,n).\texttt{provides} = p$

It is natural to define a package description as a function since we can have at most one package description for a given pair of package name and version in a CUDF document. The function is generally only partial since we clearly do not require to have a package description for any possible pair of package name and version.

We define the removal operation of a particular versioned package from a package description. This operation will be needed later in Definition 13 to define the semantics of *package conflicts* in case a package conflicts with itself or a feature provided by the same package.

Definition 3 (Package removal)**.** *Let ϕ be a package description, $p \in \mathcal{V}(\texttt{ident})$ and $n \in \mathcal{V}(\texttt{posint})$. The package description $\phi - (p,n)$ is defined by*

$$\begin{aligned} Dom(\phi - (p,n)) &= Dom(\phi) - \{(p,n)\} \\ (\phi - (p,n))(q,m) &= \phi(q,m) \qquad \text{for all } (q,m) \in Dom(\phi - (p,n)) \end{aligned}$$

Definition 4 (Request)**.** *A* request *is a triple (l_i, l_u, l_d) with $l_i, l_u, l_d \in \mathcal{V}(\texttt{vpkglist})$.*

In a triple (l_i, l_u, l_d), l_i is the list of packages to be installed, l_u the list of packages to be updated, and l_d the list of packages to be deleted.

Appendix B.2. Installations

Definition 5 (Installation)**.** *An* installation *is a function from $\mathcal{V}(\texttt{ident})$ to $P(\mathcal{V}(\texttt{posint}))$.*

The idea behind this definition is that the function describing an installation associates the set of versions that are installed to any possible package name. This set is empty when no version of the package is installed.

We can extract an installation from any package description as follows:

Definition 6 (Current installation)**.** *Let ϕ be a package description, the* current package installation *of ϕ*

$$i_\phi \colon \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$i_\phi(p) := \{n \in \mathcal{V}(\texttt{posint}) \mid (p,n) \in Dom(\phi) \text{ and } \phi(p,n).\texttt{installed} = \texttt{true}\}$$

A package can declare zero or more *features* that it provides. The function f_ϕ defined below associates to any package name (here intended to be a the name of a virtual package) the set of version numbers with which this virtual package is provided by some of the packages installed by ϕ:

Definition 7 (Current features)**.** *Let ϕ be a package description, the* current features of ϕ

$$f_\phi : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$\begin{aligned} f_\phi(p) \quad := \{ & n \in \mathcal{V}(\texttt{posint}) \mid \textit{exists } q \in Dom(i_\phi) \textit{ exists } m \in i_\phi(q) \textit{ such that} \\ & (((=, n), p) \in \phi(q, m).\texttt{provides} \textit{ or } (\top, p) \in \phi(q, m).\texttt{provides})\} \end{aligned}$$

The second case in the definition above expresses the fact that providing a feature without a version number means providing that feature at any possible version.

In order to define the semantics of a CUDF document, we will frequently need to merge two installations. This will mainly be used for merging an installation of packages with an installation of provided features. The merging operation is formalized as follows:

Definition 8 (Merging)**.** *Let $f, g : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ be two installations. Their merge $f \cup g : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ is defined as*

$$(f \cup g)(p) = f(p) \cup g(p) \qquad \textit{for any } p \in \mathcal{V}(\texttt{ident})$$

Appendix B.3. Consistent package descriptions

We define what it means for an installation to satisfy a constraint:

Definition 9 (Constraint satisfaction)**.** *The* satisfaction *relation between a natural number n and a constraint $c \in$ Constraints, noted $n \models c$, is defined as follows:*

$n \models \top$	*for any* n	$n \models (<, v)$	*iff* $n < v$
$n \models (=, v)$	*iff* $n = v$	$n \models (>, v)$	*iff* $n > v$
$n \models (\neq, v)$	*iff* $n \neq v$	$n \models (\leq, v)$	*iff* $n \leq v$
		$n \models (\geq, v)$	*iff* $n \geq v$

Now we can define what it implies for a package installation to satisfy some formula:

Definition 10 (Formula satisfaction)**.** *The* satisfaction *relation between an installation I and a formula p, noted $I \models p$, is defined by induction on the structure of p:*

- *$I \models (c, p)$ where, $c \in$ Constraints and $p \in \mathcal{V}(\texttt{ident})$, iff there exists an $n \in I(p)$ such that $n \models c$.*
- *$I \models \phi_1 \wedge \ldots \wedge \phi_n$ iff $I \models \phi_i$ for all $1 \leq i \leq n$.*

- $I \models \phi_1 \vee \ldots \vee \phi_n$ *iff there is an* i *with* $1 \leq i \leq n$ *and* $I \models \phi_i$.

We can now lift the satisfaction relation to sets of packages:

Definition 11. *Let* I *be an installation, and* $l \in \mathcal{V}(\texttt{vpkglist})$. *Then* $I \models l$ *if for any* $(c, p) \in l$ *there exists* $n \in I(p)$ *with* $n \models c$.

Note that, given that $\mathcal{V}(\texttt{veqpkglist}) \subseteq \mathcal{V}(\texttt{vpkglist})$, this also defines the satisfaction relation for elements of $\mathcal{V}(\texttt{veqpkglist})$. Also note that one could transform any $l \in \mathcal{V}(\texttt{vpkglist})$ into a formula $l_\wedge \in \mathcal{V}(\texttt{vpkgformula})$, by constructing the conjunction of all the elements of l. The semantics of l is the same as the semantics of the formula $l_\wedge$.

Definition 12 (Disjointness)**.** *The* disjointness *relation between an installation* I *and a set* $l \in \mathcal{V}(\texttt{vpkglist})$ *of packages possibly with version constraints, is defined as:* $I \parallel l$ *if for any* $(c, p) \in l$ *and all* $n \in I(p)$ *we have that* $n \not\models c$.

Definition 13. *A package description* ϕ *is* consistent *if for every package* $p \in \mathcal{V}(\texttt{ident})$ *and* $n \in i_\phi(p)$ *we have that*

1. $i_\phi \cup f_\phi \models \phi(p, n).\texttt{depends}$
2. $i_{\phi-(p,n)} \cup f_{\phi-(p,n)} \parallel \phi(p, n).\texttt{conflicts}$

In the above definition, the first clause corresponds to the *Abundance* property of [18]: all the dependency relations of all installed packages must be satisfied. The second clause corresponds to the *Peace* property of [18]. In addition, we now have to take special care of packages that conflict with themselves, or that provide a feature and at the same time conflict with that feature: we only require that there be no conflict with any *other* installed package and with any feature provided by some *other* package

Appendix B.4. Semantics of requests

The semantics of a request is defined as a relation between package descriptions. The idea is that two package descriptions ϕ_1 and ϕ_2 are in the relation defined by the request r if there exists a transformation from ϕ_1 to ϕ_2 that satisfies r.

First we define the notion of a successor of a package description:

Definition 14 (Successor relation)**.** *A package description* ϕ_2 *is called a* successor *of a package description* ϕ_1, *noted* $\phi_1 \rightarrowtail \phi_2$, *if*

1. $Dom(\phi_1) = Dom(\phi_2)$
2. *For all* $p \in \mathcal{V}(\texttt{ident})$ *and* $n \in \mathcal{V}(\texttt{posint})$*: if* $\phi_1(p, n) = (i_1, k_1, d_1, c_1, p_1)$ *and* $\phi_2(p, n) = (i_2, k_2, d_2, c_2, p_2)$ *then* $k_1 = k_2$, $d_1 = d_2$, $c_1 = c_2$, *and* $p_1 = p_2$.
3. *For all* $p \in \mathcal{V}(\texttt{ident})$
   - *for all* $n \in i_{\phi_1}(p)$*: if* $\phi_1(p, n).\texttt{keep} = \texttt{version}$ *then* $n \in i_{\phi_2}(p)$.
   - *if there is an* $n \in i_{\phi_1}(p)$ *with* $\phi_1(p, n).\texttt{keep} = \texttt{package}$ *then* $i_{\phi_2}(p) \neq \emptyset$

- *for all* $n \in i_{\phi_1}(p)$*: if* $\phi_1(p,n).\texttt{keep} = \texttt{feature}$ *then* $i_{\phi_2} \cup f_{\phi_2} \models \phi_1(p,n).\texttt{provides}$

The first and the second item of the above definitions indicate that a successor of a package description ϕ may differ from ϕ only in the status of packages. The third item refines this even further depending on keep values:

- If we have a keep status of `version` for an installed package p and version n then we have to keep that package and version.
- If we have a keep status of `package` for some installed version of a package p then the successor must have at least one version of that package installed.
- If we have a keep status of `feature` for some installed version n of a package p then the successor must provide all the features that where provided by version n of package p.

Definition 15 (Request semantics)**.** *Let* $r = (l_i, l_u, l_d)$ *be a request. The* semantics *of* r *is a relation* $\overset{r}{\rightharpoonup} \subseteq \textsc{Descr} \times \textsc{Descr}$ *defined by* $\phi_1 \overset{r}{\rightharpoonup} \phi_2$ *if*

1. $\phi_1 \rightarrowtail \phi_2$
2. ϕ_2 *is consistent*
3. $i_{\phi_2} \cup f_{\phi_2} \models l_i$
4. $i_{\phi_2} \cup f_{\phi_2} \parallel l_d$
5. $i_{\phi_2} \cup f_{\phi_2} \models l_u$*, and for all* p *such that* $(c,p) \in l_u$ *we have that* $(i_{\phi_2} \cup f_{\phi_2})(p) = \{n\}$ *(i.e., is a singleton set) where* $n \geq n'$ *for all* $n' \in (i_{\phi_1} \cup f_{\phi_1})(p)$*.*

Appendix C. Translating Debian package metadata to CUDF

In this section we provide the details of the translation of Debian package metadata to CUDF. Debian semantics is carefully described and documented in the Debian Policy [12] and enforced by various quality assurance tools currently used by Debian contributors. However, despite the thoroughness of the Debian Policy, this offers only an informal description.

Appendix C.1. Debian metadata

Debian package metadata is stored in different files on user machines. These files are handled by the low level package installer `dpkg` and the package manager `apt-get`. Data that is kept up to date by the package installer are related to the actual state of the machine while packages maintained by the package manager are mainly—but not only—information related to all available packages available in Debian repositories. The *status* of the machine consists of a snapshot of the current state of the installer.

The package *universe* is the set of all packages known to the package manager. The universe for an upgrade problem is available on a machine as the lists

of packages known to `apt`. These files are downloaded when the `update` action is performed, and constitute a local cache of package listings coming from the Debian archive or its mirrors around the world.

There are many other files used to specify specific parameters of the low and high level package manager as well as to specify remote resources used by the high level package managers to retrieve binary packages from the net. These are not interesting in this context.

Appendix C.2. Mapping Debian metadata to CUDF

To produce a valid CUDF document from a Debian universe and status we need to address both syntactic and semantic differences. In particular, the conversion of the Debian universe involves the following three steps:

1. version and package name normalization,
2. addition of self conflicts,
3. virtual package normalization, and
4. marking of the installed packages

The overall complexity of this transformation is superlinear with a low coefficient in the number of packages in the universe.

Version and name normalization. The CUDF specification [24] requires version numbers to be integers. In order to normalize Debian versions, it is necessary to collect for each package all versions mentioned in the document (including in conflicts and dependencies), and to sort these versions in ascending order according to the Debian version comparison function (see Section 5.6.12 of [12]). Then, the versions are mapped to integers $1, 2, \ldots$, so that the order is preserved.

Furthermore, CUDF syntax also requires a normalization on package names by escaping characters that are not permitted by the CUDF specification.

Self conflicts. Debian semantics does not allow more than one version of the same package to be installed at the same time. In order to make this constraint explicit in the CUDF document, we add to each package a self conflict without a version.

This implies that we have to rename any virtual package that carries the same name as a concrete package. Without this renaming, installation of a package providing some virtual package p together with a concrete package p would be artificially excluded by package p conflicting with p.

Example 1. *Consider the Debian packages in Figure C.5(a).*

In Debian semantics, package `foo` *(in either version 1 or 2) can be installed together with package* `bar`*. Exactly the same file, but interpreted in CUDF semantics, would allow to install versions 1 and 2 of* `foo` *at the same time, which is not possible in Debian. We hence have both versions of* `foo` *conflict with* `foo` *in order to retrieve the mutual exclusion between both versions of* `foo` *(Figure C.5(b))*

```
Package: foo
Version: 1

Package: foo
Version: 2

Package: bar
Version: 42
Provides: foo
```

(a)

```
package: foo
version: 1
conflicts: foo

package: foo
version: 2
conflicts: foo

package: bar
version: 42
provides: foo
```

(b)

```
package: foo
version: 1
conflicts: foo

package: foo
version: 2
conflicts: foo

package: bar
version: 42
provides: foo--virtual
```

(c)

Figure C.5: Two attempts to translate Debian metadata (a) into CUDF (b), (c).

This renaming has the unwanted side-effect that now `foo` *and* `bar` *are in conflict. We hence rename the virtual package provided by* `bar`*, and obtain finally the correct encoding in Figure C.5(c).*

In the following we will see a different case where virtual packages must be renamed.

Virtual package normalization. In Debian, a package can be both a virtual package and a concrete package. *Dependencies* and *conflicts* on virtual packages possibly carry a version constraint (see Section 7.5 of [12]). If they do not carry a version constraint then both virtual packages and concrete packages may be used to satisfy a dependency, and both are relevant for conflicts. If a relation carries a version constraint then only concrete packages are relevant. On the other hand, *Provides* in Debian do not carry a version, and can never satisfy a dependency on a package that carries a version constraint. In CUDF, however, provides without a version constraint are quantified universally over all available versions.

Example 2. *Consider the Debian packages in Figure C.6(a).*

The package `foo` *has a constrained dependency on* `baz` *which is in turn provided by the package* `extra`*. In Debian, this dependency can only be satisfied by the concrete package* `baz` *and not by the concrete package* `extra` *since the virtual package* `baz` *provided by* `extra` *cannot satisfy a dependency with a version constraint. If we were to propose a naive translation from the fragment above to CUDF then there would be a mismatch between the Debian and the CUDF semantics. In particular, in the latter case, the package* `extra` *would allow to satisfy* `foo`*'s dependencies, while in Debian this is not the case.*

In order to reconcile the two different semantics of Debian and CUDF we perform the following translation:

- All package names p in some `provides` property for which there exists a concrete package with the same name or a version constraint associated to it are replaced by a new package name p`--virtual`.

```
Package: foo
Version: 1
Depends: baz (>= 2)

Package: bar
Version: 1
Depends: baz

Package: baz
Version: 2

Package: extra
Version: 2
Provides: baz
```

(a)

```
package: foo
version: 1
conflicts: foo
depends: baz >= 2

package: bar
version: 1
conflicts: bar
depends: baz | baz--virtual

package: baz
version: 2
conflicts: baz

package: extra
version: 2
conflicts: extra
provides: baz--virtual
```

(b)

Figure C.6: (Correctly) translating dependencies of Debian packages (a) into CUDF (b).

- All relations without a version constraint to a package that exists in the CUDF document as virtual package are expanded to an alternative consisting of all matching concrete or (renamed) virtual packages.
- All relations without a version constraint that match only concrete packages in the CUDF document are left untouched, since for concrete packages the Debian semantics and CUDF semantics coincide.
- All relations carrying a version constraint are left untouched as they are not going to match a virtual package anyway (since these have been renamed)

For instance, consider the following Debian packages in Figure C.6(a) where the package `baz` is both a concrete package and a virtual package (provided by the package `extra`).

The corresponding CUDF document is in Figure C.6(b), where the virtual package `baz` is replaced by a new package name `baz--virtual` and the dependency of the package `baz` is replaced by a disjunction as it is not constrained.

Note that in this example, using the Debian semantics, package `baz` and package `extra` can be installed together. Without the renaming packages `baz` and `extra` could not be installed together in CUDF because of the self conflict.

A concrete example of the conversion of a Debian package description to CUDF format is as follows :

```
Package: 6tunnel
Priority: optional
```

```
Section: net
Installed-Size: 68
Maintainer: Thomas Seyrat <tomasera@debian.org>
Architecture: i386
Version: 0.11rc2-2
Depends: libc6 (>= 2.3.6-6)
Filename: pool/main/6/6tunnel/6tunnel_0.11rc2-2_i386.deb
Size: 12810
MD5sum: 5471e156d43755878763ec51a86ac1aa
SHA1: 8af63219150ad7079e5fb412c37b0b8e78904159
SHA256: 192db6cede7fc2794bccc6662b29f6935e84a59bb5cbf64b15989d114bc15c8a
Description: TCP proxy for non-IPv6 applications
```

The CUDF conversion results in the following stanza where additional fields not relevant to CUDF have been omitted:

```
package: x6tunnel
version: 0
depends: libc6 >= 1
conflicts: x6tunnel
```

Appendix D. Translating RPM package metadata to CUDF

This section describes the translation from a RPM package archive to CUDF. The RPM semantics is not formally described, but it is supposed to be consistent with the latest implementation of the RPM utilities. In this document we refer to the implementation of `librpm` version 4.4.2.2.

Appendix D.1. Version expansion

RPM versions are triples of the form `epoch, version, release`. The *epoch* term is an integer and it is used to allow to replace new RPM packages where RPM considers the new package version number to be lower than the installed package. The default epoch is zero and it is usually not specified. The *version* term is a sequence of alpha-numeric characters identifying the upstream version of the package. The *release* term is a sequence of alpha-numeric characters commonly identifying a distribution-specific release code. RPM versions identify *concrete* packages as a triple, while it is written as a string of the form `epoch:version-release` when used in dependency information items. In order to normalize RPM versions into a common format, we rewrite all RPM versions as strings.

Consider the following representation of a RPM package:

```
package: bash
version: 1.3
epoch: 1
release: ex2010
```

We associate to this package the canonical version string `1:1.3-ex2010`. If the epoch is not specified then it defaults to 0.

Algorithm 1 RPM version comparison

```
function VERCMP(⟨e1, v1, r1⟩, ⟨e2, v2, r2⟩)
    re ← epochcmp(e1, e2)
    if re = 0 then
        rv ← rpmvercmp(v1, v2)
        if rv = 0 then
            return relcmp(r1, r2)
        else
            return rv
        end if
    else
        return re
    end if
end function
```

Appendix D.2. RPM comparison function

RPM versions are compared using four functions: `vercmp` (shown in Algorithm 1), `epochcmp`, `relcmp` and `rpmvercmp`. The function `vercmp` compares two RPM triples $\langle epoch, version, release \rangle$. First we compare the epoch using `epochcmp`. If these are equal then we compare the versions using `rpmvercmp`, and if these are equal again we compare the releases using `relcmp`.

`rpmvercmp` is the RPM comparison function as implemented in the RPM library. Since the RPM comparison function has no normative specification, we do not describe its algorithm here.

The `epochcmp` function compares the two epoch if they are present, and returns 1 or -1 if only one is present, 0 otherwise. Similarly, the `relcmp` function compares the two releases if they are both present, and return 1 or -1 if only one is present, 0 otherwise.

Appendix D.3. CUDF version mapping

CUDF versions are strictly positive integers [24]. RPM versions, which are strings, must hence be mapped from string to integer. However, the naive approach of having a bijective mapping does not work since the RPM comparison function above does not provide a total order of RPM versions. To overcome this problem, the mapping from rpm versions to CUDF version is done by explicitly computing all constraints during the translation phase using, hence providing an explicit (and specific) ordering of all considered rpm versions. A second phase assigns an integer to each rpm version using the order induced by this procedure.

Appendix D.4. Dependency mapping

Since all version constraints are fully expanded in the CUDF document, the dependencies translation is straightforward. There are two minor points to notice. The first point is that we remove dependencies on files provided by the

same package, as well as dependencies on the package itself and dependencies on packages provided by the same package. This reduces the size of the resulting CUDF document and removes redundant information. The second point concerns relations to non-existing packages. In order to remain faithful to the CUDF mission, hence to retain all information contained in the source format, versions are numbered starting from 2 while non-existing packages are numbered 1. This information is available at the conversion stage because of the constraint expansion.

The mapping algorithm works in two stages. In the first stage, we build a unit table which maps package names to arrays of constraints. The indices of the array will be used as CUDF versions.

In the second stage, for each dependency expansion, a constraint of the form `(name,flag,version)` will be matched against the list of constraints in the units table. If the constraint overlaps the constraint in the units table then the associated index is added to the dependency list in the form `(name,"=",cudf version)`. Otherwise it is ignored.