MPM : a modular package manager*

Pietro Abate
abate@pps.jussieu.fr

Roberto Di Cosmo
roberto@dicosmo.org
INRIA, France

Ralf Treinen
treinen@pps.jussieu.fr

Stefano Zacchiroli
zack@pps.jussieu.fr

Université Paris Diderot, PPS, UMR 7126, Paris, France

ABSTRACT

Software distributions in the FOSS world rely on so-called *package managers* for the installation and removal of packages on target machines. State-of-the-art package managers are monolithic in architecture, and each of them is hard-wired to an ad-hoc *dependency solver* implementing a customized heuristics. In this paper we propose a modular architecture allowing for pluggable 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, accept more expressive input languages, and can be easily adaptable to new platforms. We present a working prototype—called MPM—which has been implemented following the design advocated in this paper.

Primary Classification:

D. Software

D.2 SOFTWARE ENGINEERING

D.2.11 Software Architectures

Subjects: Domain-specific architectures

General Terms: Algorithms, Design, Experimentation, Measurement
Keywords: package managers

1. INTRODUCTION

Free and Open Source Software (FOSS) distributions, as well as other complex software platforms, strive to provide modular software *components* (or packages) that can be assembled to provide the user with the desired functionalities. Packages are described by stanzas of meta-information; these include complex relationships to other packages describing the requirements for a package to run properly on a target system. So-called *package managers* are used to perform installation, upgrade and removal of packages on target machines. Package managers incorporate numerous functionalities: (trusted) retrieval of components from remote repositories; planning of upgrade paths that respect inter-component constraints (also known as *dependency solving*); user interaction to allow for fine-tuning of the choice of components; and finally the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered.

Until recently, package managers in FOSS distributions followed a monolithic architecture (re-)implementing all functionalities to fit specific formats of meta-data 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 [6] that some of the authors have shown that for packages in GNU/Linux 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 [13] have pushed various communities to incorporate SAT solvers directly in package managers, instead of writing ad-hoc solvers as it was previously the case [4, 10, 17, 18].

In this paper 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 based on two domain specific languages (DSL). 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 Debian, in terms of quality of the proposed solution.

*Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project.
Work performed at the IRILL center for Free Software Research and Innovation in Paris, France.

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

The rest of the paper is structured as follows: Section 2 introduces the package installation problem. We discuss the state of the art in package solvers used in FOSS distributions, we give a concrete example of such problems and show the solution proposed by MPM. In Section 3, we discuss the design of a modular package manager and we present two domain specific languages. Section 4 introduces a 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, Section 6 discusses related research.

2. THE UPGRADE PROBLEM

A stable GNU/Linux distribution nowadays undergoes a strict quality assurance process, and contains only one version of each package[1]. As a consequence, the initial installation of a GNU/Linux system from scratch usually runs little risk of producing dependency problems. Installing, removing, and upgrading components in such configuration is a task that is 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 "baselines") 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 into 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 is requested to guide interactively the package manager, by suggesting choices among packages which may seem totally unrelated to the original request.

2.1 State of the Art

FOSS distributions are mainly divided into RPM and Debian based distributions. This division is imposed by different package formats, and consequently different low-level package managers (`rpm` versus `dpkg`) and different formats of metadata. 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 SAT solver (`libzypp`), but preferences are hard coded in the problem encoding [19, 10]. `apt-pbo` is a fork of apt-get and uses a pseudo-Boolean solver as dependency solving engine. It is possible to specify custom optimization functions to reflect user preferences [17].

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 [12] in its latest revision implements an experimental CUDF (see Section 3.1) back-end. `apt-get` and `aptitude` are both based on libapt, which is a low-level library used to handle user configuration files, package retrieval, and installation planning. Both `apt-get` and `aptitude` contain an ad-hoc dependency solver. Despite performing very well on a stable release of the debian distribution, it shows its shortcoming when mixing repositories of different debian releases, and when dealing with unusually complicated scenarios ([7], see also Section 2.2). `apt-get` has an experimental support to use an external SAT solver, however it is still not possible to specify user criteria directly [9].

One exception to this classification is the cross distribution package manager `smart`. It has a modular architecture w.r.t. the package installer backend and the meta-data 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, w.r.t. some criteria, is computationally hard and is treated differently by different tools. Some rely on special heuristics, like Apt 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 having unacceptably high computation times on some cases; as we have seen, the solution found is not necessarily always optimal either.

2.2 A problematic example

In this section, we present 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 two consecutive stable Debian releases, which have passed strict quality assurance tests.

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

We tested apt-get, aptitude, cupt and smart, whose default behaviour is to try and install the latest available version, that 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, apt-get at the same time requires to remove 414 components to proceed, which is clearly unacceptable.

```
apt-get --without-recommends install python-simpy
```

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

```
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
```

2.3 Using the Mancoosi Package Manager

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 an installation 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 in 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 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 problem. Re-developing from scratch dependency solvers as soon as dependencies and conflicts are introduced in yet another component model seem 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 [sic] of dependency solving from that of specific package managers *and* component models.

To attain this goal we have designed a modular architecture based on two domain specific languages; the first is a common format for describing package upgrade problems, CUDF that is both distribution-agnostic and neutral w.r.t. the solving technology; the second is a simple, yet powerful optimization language apt to specify common user requirements and to select an optimal solution.

3.1 Common Upgradeability Description Format

CUDF (for Common Upgradeability Description Format) is a specialised language used to describe upgrade problem instances, that is presented in detail in [16]. Here we just recall that it has been designed with the following goals in mind:

Platform independence.

We want 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.

Solver independence.

In contrast to encodings of inter-component relations which are targeted at specific solver techniques, CUDF stays close to the original problem, in order to preserve its structure and avoid bias towards specific solvers.

Readability.

CUDF is a compact plain text format which makes it easy for humans to read upgrade scenarios, and facilitates interoperability with package managers.

Extensibility.

Only core component properties that are shared by mainstream platforms and 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.

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.

Figure 1 shows the concrete syntax of CUDF document in a sample upgrade scenario.

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

package: car
version: 1
depends: engine, wheel, door, battery
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 > 2
```

Figure 1: Sample CUDF document.

3.2 Optimization criteria

The second DSL that we present is a simple language to specify optimization criteria.

It is easy to remark that a user request may have exponentially many solutions when many components are installable independently. Hence, it is necessary to allow users to express their preferences about the solutions they really look for.

The state-of-the-art approach in FOSS package managers 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 change the solution.

We propose an alternative approach which consists in letting the user specify high-level criteria to 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". Our proposal for a high-level user preference language is simple yet expressive:

1. define a dictionary of useful criteria: our prototype supports, among others, *removed*, *changed*, *new*, *notuptodate*, which are self-explanatory;

2. define a dictionary of aggregation functions lex, $leximin$, $leximax$, etc. Our prototypes assumes by default a lexicographic ordering of the criteria;

3. write the user preference as an expression $op(\circ c_1, \ldots, \circ c_n)$ where op is an aggregation function, $\circ \in \{+, -\}$ (for maximize/minimize, resp.), and the c_i are criteria.

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.

This is the optimization function we used for the benchmarks reported in the next section, but it is straightforward

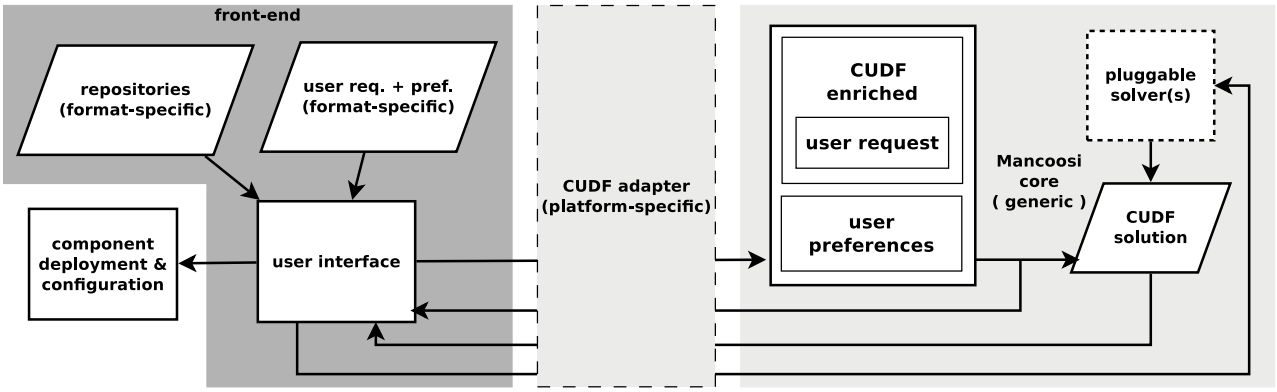

Figure 2: Modular package manager architecture

to write many others criteria, like

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

which is a criterion capturing the preferences of a user who wants to have the latest version of each package installed, as far as no functionality is removed, and without installing more packages than required.

3.3 A modular architecture

The proposed modular architecture is shown in Figure 2. 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. This architecture brings back separation of concerns to package management design:

Format conversion.
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;

Dependency 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;

Package manager.
developers may focus on the killer feature of their software (trust management, user interface and interaction, transactional upgrade deployment, etc.) and stop worrying about solver issues.

In terms of architectural constraints, we notice that our proposed architecture makes no assumptions on the distribution of the different elements that make it up: of course, a distributed implementation would require to address reliability and fault tolerance issues more deeply than a centralised implementation, which are out of the scope of this paper.

In the next section we propose a centralised implementation of the architecture described in Figure 2.

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 solver APT 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>`
- 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 complicated 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 Picking a dependency solver

Any dependency solver that uses CUDF as input and output format, and that accepts the optimization criterion language described in Section 3.2 can be used in conjunction with MPM. CUDF is precisely the input format of the *Mancoosi International Solver Competition (MISC)* [14], 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 MISC competition 2010, and use CUDF as input and output format:

1. `apt-pbo` [17], using a PBO solver
2. `aspcud`, based on Answer Set Programming using Potassco, the Potsdam Answer Set Solving Collection
3. `inesc` [2], using the p2cudf parser (from Eclipse) and the MaxSAT solver MSUnCore
4. `p2cudf` [3, 4], 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` [15], based on ILOG's CPLEX

We expect the list of competition participants to grow for the 2011 issue of the MISC competition. In november 2010, three of these solvers plus a new one which has entered since MISC 2010, already accept the user preference language, and are hence eligible as solver plugin to MPM. The current prototype implementation of MPM employs `aspcud`.

4.3 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` [1] to translate debian package metadata to CUDF; translators to CUDF are also available for the meta-data format of `rpm` [1] and Eclipse [5]. The back translation of the solution, expressed in CUDF, to `apt` is performed by the `cudf_sol_diff` tool that computes the difference between the initial package status and the one found in the solution.

MPM is written in python and it 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.

The command line interface of MPM is a super-set of apt (see the `apt-get` (1) man page). In addition, the user can 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 a "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

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 Mancoosi International Solver Competition (MISC 2010) [14].

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 on line at `http://data.mancoosi.org/papers/cbse2011`.

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 (v. 0.8.10), aptitude (v 0.6.3), smart (v. 1.3-1), and cupt (v. 1.5.14.1). The first two package managers are the most representative, as they are the standard ones on Debian; the last two were selected because available at the time of writing in the Debian distribution and at the same time capable

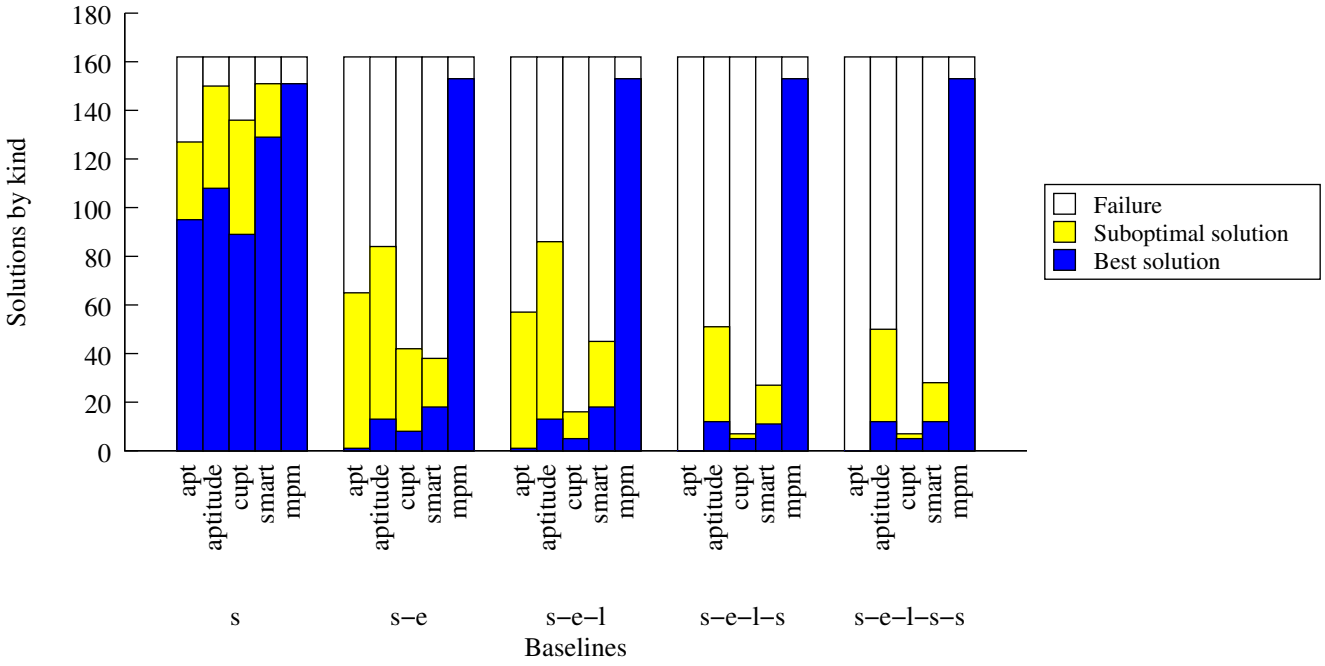

Figure 3: Comparison of mainstream meta-installer with MPM on increasingly large package universes. Baselines are formed taking increasingly large sets of Debian releases, indexed by the first letter of their name: sarge, etch, lenny, squeeze, sid.

of working with Debian meta-data. All package managers were used with their default options.

All these legacy solvers have hardwired some kind of optimization criterion, so it is not easy 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 w.r.t. 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 [14], 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.

For each of these five universes, we have run 162 user requests, half of them requiring the installation of 5 packages and half requiring the removal of 5 packages. Package managers were given a timeout of 60 seconds for each request.

Assessment.

In Figure 3 we show the aggregate results of our tests where solutions are ranked according to the "paranoid" criterion and then divided into three categories, *optimal*, *sub-optimal* and *failure*. Optimal and sub-optimal categories hold respectively the best solutions (including ex-aequos) and not optimal, but still correct solutions. 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 25% of the cases, with the notable exception of *smart* which is almost as good as MPM.

The situation changes radically as soon as more than one baseline is considered: all legacy package manager fail to find a solution in more than 50% of the cases, and their hit rate drops considerably when adding more baselines. This is consistent with the folklore experience of GNU/Linux 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 relies on a state-of-the-art external solver, that is required to find a *global* solution that is optimal w.r.t. the paranoid criterion, unlike the legacy package managers which apply ad-hoc local search algorithms.

These results also justify the fact of not comparing execu-

tion times: when the maintenance problem become difficult, MPM is the only viable tool, even in its prototype, unoptimized form.

6. RELATED WORK

To the best of authors knowledge, the proposal of making dependency solvers a modular component of package managers is novel. In spite of that, several attempts at using specific formal techniques to attack dependency solving problems have been made in the past; in some cases they resulted into 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 [7, 13]. Following that initial encoding, the use of SAT technology in package managers has seen a raise in popularity. The OPIUM system [18] used in 2006 a SAT solver with an optimization in line with our paranoid criterion, SUSE libzypp [10] incorporated a SAT solver in 2007, the Eclipse P2 system includes the Sat4J solver since 2007 [3]. 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 [17]. 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 [13, 18, 3], Pseudo Boolean Optimization [17], Partial Weighted Max SAT [2], Mixed Integer Linear Programming [15], as well as some others championed by entrants in the MISC 2010 competition (see Section 4.2).

Jenson [8] 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 [3] where it was also observed that much of the complexity stems from multiple versions of components and the constraints they entail.

7. CONCLUSIONS

We have shown 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; it 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, MPM, for Debian-based GNU/Linux distribution, based on this architecture, by reusing 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 significanlty 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.

Acknowledgements

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 acknowledgement 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.

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/cbse2011/`. A synthetic presentation of the results can be found at `http://www.mancoosi.org/measures/packagemanagers/`.

8. REFERENCES

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

[2] J. Argelich, D. L. Berre, I. Lynce, J. Marques-Silva, and P. Rapicault. Solving Linux upgradeability problems using Boolean optimization. In Lynce and Treinen [11], pages 11–22. `http://arxiv.org/abs/1007.1021v1`.

[3] D. L. Berre and A. Parrain. On SAT technologies for dependency management and beyond. In S. Thiel and K. Pohl, editors, *12th International Conference on Software Product Lines, Second Volume (Workshops)*, pages 197–200, Limerick, Ireland, Sept. 2008. Lero Int. Science Centre, University of Limerick, Ireland.

[4] D. L. Berre and P. Rapicault. Dependency management for the Eclipse ecosystem. In R. D. Cosmo and P. Inverardi, editors, *International Workshop on Open Component Ecosystems (IWOCE 2009)*, Amsterdam, The Netherlands, 2009.

[5] C. Bozman. Converting Eclipse metadata into CUDF. Technical Report TR5, The Mancoosi Project, Nov. 2010. `http://www.mancoosi.org/reports/p2.pdf`.

[6] EDOS Project WP2 Team. First report on formal management of software dependencies. Technical Report D2.1, Sept. 2005. `http://www.edos-project.org/xwiki/bin/Main/Deliverables`.

[7] EDOS Project WP2 Team. Second report on formal management of software dependencies. Technical Report D2.2, Mar. 2006. `http://www.edos-project.org/xwiki/bin/Main/Deliverables`.

[8] G. Jenson, J. Dietrich, and H. W. Guesgen. An empirical study of the component dependency

resolution search space. In L. Grunske, R. Reussner, and F. Plasil, editors, *13th International Symposium on Component-Based Software Engineering*, volume 6092 of *Lecture Notes in Computer Science*, pages 182–199, Prague, Czech Republic, June 2010. Springer-Verlag.

[9] J. A. Klode. RFC: The future of solving dependency problems in APT (SAT, CUDF). `http://lists.debian.org/deity/2010/12/msg00053.html`.

[10] Libzypp library. `http://en.opensuse.org/Portal:Libzypp`, 2011.

[11] I. Lynce and R. Treinen, editors. *Proc. 1st International Workshop on Logics for Component Configuration*, volume 29 of *EPTCS*, 2010.

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

[13] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 199–208, Tokyo, Japan, Sept. 2006. IEEE CS Press.

[14] Mancoosi. Mancoosi international solver competition 2010. `http://www.mancoosi.org/misc-2010/`, July 2010.

[15] C. Michel and M. Rueher. Handling software upgradeability problems with MILP solvers. In Lynce and Treinen [11], pages 1–10. `http://arxiv.org/abs/1007.1020v1`.

[16] R. Treinen and S. Zacchiroli. Description of the CUDF format. Technical Report D5.1, The Mancoosi Project, Nov. 2008. `http://www.mancoosi.org/reports/tr3.pdf`.

[17] P. Trezentos, I. Lynce, and A. L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-Boolean optimization. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pages 427–436, Antwerp, Belgium, Sept. 2010. ACM.

[18] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 178–188, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[19] Yast. `http://en.opensuse.org/Package_management`, 2011.