Apt-pbo: Solving the Software Dependency Problem using Pseudo-Boolean Optimization

Paulo Trezentos
ISCTE/ADETTI/Caixa Magica
Av.Forças Armadas
Lisbon, Portugal
prrt@iscte.pt

Inês Lynce
INESC-ID/IST/TU Lisbon
Av. Rovisco Pais
Lisbon, Portugal
ines@sat.inesc-id.pt

Arlindo L. Oliveira
INESC-ID/IST/TU Lisbon
Av. Rovisco Pais
Lisbon, Portugal
aml@inesc-id.pt

ABSTRACT

The installation of software packages depends on the correct resolution of dependencies and conflicts between packages. This problem is NP-complete and, as expected, is a hard task. Moreover, today's technology still does not address this problem in an acceptable way. This paper introduces a new approach to solving the software dependency problem in a Linux environment, devising a way for solving dependencies according to available packages and user preferences. This work introduces the "apt-pbo" tool, the first publicly available tool that solves dependencies in a complete and optimal way.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]

General Terms

Algorithms,Reliability

1. INTRODUCTION

Software installation is the process of installing programs assuring that specifically required software is pre-installed and that defined actions are taken before or after the copy of the files into the file-system [24, 29]. Although this is a common problem among Microsoft and Free / Open Source Operating Systems (Linux, BSD,...) [30] we will focus on the later ones, since a progress in this field would be applicable to all environments, including applications like Eclipse or Firefox [13].

The installation process comprises retrieving the package, solving the software dependency tree, retrieving and installing the software dependencies and finally installing the package and executing the associated install scripts [6].

The dependency graph represents the software dependencies and sub-dependencies needed for a package to work properly after installation [4]. The restrictions imposed by the graph may have no solution (for instance, due to broken dependencies), only one solution, or several solutions. Criteria such as the minimum number of packages or up-to-date packages can be defined to rank the solutions in terms of their quality. Finding a solution consists in defining the sub-set of packages that meets the dependency requirements. This process is called dependency solving.

There are three main contributions in this paper. First, our main finding is an efficient encoding of the dependencies and conflicts as a pseudo-Boolean optimization problem without the need for Integer Linear Programming (ILP) or Satisfiability (SAT) extra-steps, thus providing a tool capable of finding a solution in every possible case. Second, we achieve this goal by proposing a flexible algorithm that can be adjusted to user preferences without sacrificing performance, a critical issue for a tool with user interaction. Finally, the developed tool is available under a free license, allowing enhancements and benchmarking by the research community.

This paper is organized as follows. In the next section we review the basic principles of the Dependency Solving problem as well as an introduction to Pseudo-Boolean Optimization. Section 3 describes the methodology used, providing a system overview and discussing different optimization criteria. Section 4 presents experimental results and Section 5 describes related work. Finally, section 6 presents the concluding remarks.

2. BACKGROUND

2.1 Dependency Solving Problem

The dependency solving was proved to be NP-complete [14]. This confirms a well known relationship: an increase in the number of packages causes a rapid increase in the complexity of the problem, for which no efficient solutions are known.

Finding a solution becomes rapidly more difficult as the number of available packages grows and the number of versions of each package increases [20]. Existing installers like Apt, YUM, Smart or URPMI are known to report no solution even if one exists, since the search performed is incomplete.

2.1.1 *Problem Definition*

A package is installable if all of its package dependencies can be satisfied using packages in active repositories (repositories being a collection of packages that are commonly stored remotely). The number of dependencies and

ASE'10, September 20–24, 2010, Antwerp, Belgium.
Copyright 2010 ACM 978-1-4503-0116-9/10/09 ...$10.00.

conflicts between packages can become very high in the scope of open source repositories of software [11].

Although Open Source Systems (OSS) and namely Linux distributions share the basic software components (glibc, gcc,...), the packaging system is slightly different amongst them. Some distributions use the RPM format (Red Hat Package Management system), others use the DEB format (Debian system) and a minority use the almost deprecated Tarball system. The dependency solving problem applies equally to RPM and DEB, given that the existing differences are in format specification and do not compromise a universal definition of the problem.

The following definitions are needed to define the Dependency Solving problem [5, 14]:

- **Package**: A package p is a pair (u, v) where u is a unit and v is a version. Units are represented by strings and versions are represented by non-negative numbers. The set of all packages in a repository is denoted by $\mathcal{P}$.

- **Dependencies**: A dependency[1] of a package p is a set of sets of packages $d_1, \ldots, d_k$. A dependency may include a package version constraint defined by one of the following binary operators: $<, \leq, \geq, >$ or $=$. Such a dependency corresponds to a set of dependencies, each of which is called sub-dependency. If p is to be installed, then each dependency d_i needs to be satisfied, i.e. at least one package in each set of packages d_i must be installed.

- **Conflicts**: A conflict is a symmetric relation between two packages. Two packages conflict with each other if that is stated in their metadata information, if they have an element e with exactly the same location in the file system or if they are different versions of the same package [2].

- **Repository**: A repository can be represented as a tuple $\mathcal{R} = (\mathcal{P}, \mathcal{D}, \mathcal{C})$, where $\mathcal{P}$ is the set of all packages, $\mathcal{D} : \mathcal{P} \rightarrow \wp(\mathcal{P}))$ is the dependency function (with $\wp(X)$ representing the set of subsets of X) and $\mathcal{C} : \mathcal{P} \times \mathcal{P}$ is the conflict relation.

- **Dependency tree**: A dependency tree $\mathcal{DT}$ of a package p is an AND/OR graph $\mathcal{G}$ where each node corresponds to a package or a set of packages and the edges correspond to the (sub-)dependencies. The root of the tree corresponds to package p. The dependencies are represented with AND edges and the sub-dependencies are represented with OR edges. Moreover, conflicts may also be represented.

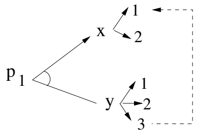

Figure 1: Generic dependency tree

Figure 1 depicts a set of packages $\mathcal{P}$ that under repository $\mathcal{R}$ has dependencies, sub-dependencies and conflicts[3]. The partial representation of repository $\mathcal{R}$ in this figure may by defined by the tuple $(\mathcal{P}, \mathcal{D}, \mathcal{C})$ with:

- $\mathcal{P} = \{(p, 1), (x, 1), (x, 2), (y, 1), (y, 2), (y, 3)\}$

- $\mathcal{D}((p, 1)) = \{\{(x, 1), (x, 2)\}, \{(y, 1), (y, 2), (y, 3)\}\}$ $\mathcal{D}((x, 1)) = \mathcal{D}((x, 2)) = \mathcal{D}((y, 1)) = \mathcal{D}((y, 2)) = = \mathcal{D}((y, 3)) = \{\}$

- $\mathcal{C}((y, 3)) = (x, 1)$

  Empty sub-sets for the dependencies mean that the package itself meets the dependency requirements.

This concept can be applied to a specific example of a package installation. For this purpose we present the dependency tree for the *car* package in Figure 2. The *car* package depends on packages *engine*, *wheel* and *door*.

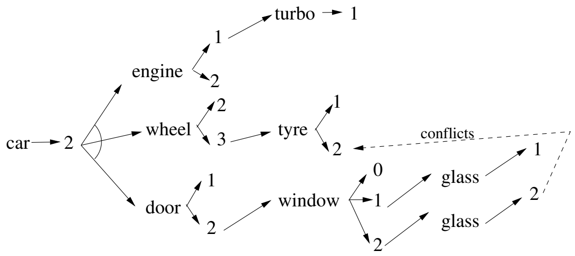

Figure 2: Dependency tree for the *car* package

A successful dependency solving algorithm for package p should derive the correct set of packages from the corresponding dependency tree $\mathcal{DT}_p$. Since, in most cases, several solutions can be found, the algorithm should derive the solution that best meets user preferences. In section 3.2 we introduce criteria for solution assessment.

2.2 Boolean Satisfiability and Pseudo-Boolean Optimization

The dependency solving problem can be easily encoded into Boolean satisfiability (SAT) using the Conjunctive Normal Form (CNF), where a CNF formula is a conjunction ($\wedge$) of clauses, a clause is a disjunction ($\vee$) of literals and a literal is a Boolean variable (x) or its negation ($\bar{x}$).

Each package is represented by a Boolean variable and, therefore, a variable assigned value true corresponds to an installed package, whereas a variable assigned value false corresponds to a package that is not installed. In addition, dependencies and conflicts are represented by clauses. For each package p and its dependencies $d_1, \ldots, d_k$, are created k clauses: $(\bar{p} \vee d_1)$, ..., $(\bar{p} \vee d_k)$. Moreover, a binary clause $(\bar{p} \vee \bar{p}')$ is created for each conflict between two packages p and p'.

For example, the dependency tree in Figure 1 is represented by the following CNF formula:

$$(\bar{p}_1 \vee x_1 \vee x_2) \wedge (\bar{p}_1 \vee y_1 \vee y_2 \vee y_3) \wedge (\bar{x}_1 \vee \bar{y}_3)$$

[1] Dependencies are defined as *Requires* in RPM terminology.

[2] By definition in DEB and by practice with exceptions in RPM.

[3] For simplicity, the tuple representation of a package as $(p, 2)$ will be now represented as p_2

However, it should be noted that SAT is a decision problem and any solution to the problem is equally valid. In case some solutions are better than the others, we move to an optimization problem and a extension of the SAT formulation called pseudo-Boolean optimization (PBO) [22] should be used instead.

In a pseudo-Boolean formula, variables have Boolean domains and constraints (called PB-constraints) are linear inequalities with integer coefficients,

$$\sum_i c_i \cdot l_i \geq n \quad \text{with } i \in \mathbb{N}, c_i, n \in \mathbb{Z} \text{ and } l_i \in \{0,1\}, \qquad (1)$$

where c_i and n are integer constants and l_i are literals. The operator $\geq$ may be replaced by other operators such as $>, =, <, \leq$.

From an Integer Linear Programming (ILP) point of view, PB-constraints can be seen as a specialization of ILP where all variables are Boolean, which explains why this formulation is also known as 0-1 ILP. From a SAT point of view, PB-constraints can be seen as a generalization of clauses. For example, clause $(\bar{p}_1 \vee x_1 \vee x_2)$ is translated to the PB-constraint $\bar{p}_1 + x_1 + x_2 \geq 1$. If one considers that $\bar{p}_1$ can be replaced by $(1 - p_1)$ then we obtain the equivalent PB-constraint $x_1 + x_2 - p_1 \geq 0$

A pseudo-Boolean formula can be extended with an optimization function, in which case the resulting formula is called a PBO formula. In the context of the dependency solving problem, the optimization function can be used to establish preferences between packages. Clearly, some installed packages are more important than others, and therefore guaranteeing that these packages are not uninstalled should be a priority.

3. METHODOLOGY

In this section we present our methodology for using pseudo-Boolean optimization for solving dependencies and conflicts in a package installation.

3.1 System Overview

Figure 3 depicts a typical installation flow of apt-pbo.

The core of the system is the *apt-pbo* application [26] that has different hooks to integrate modules. However in the tests performed, the overhead of the external calls is not significant since the number of iterations is extremely low. Note that this architecture allows the exchange of modules. For example, changing the PBO solver is an extremely easy task.

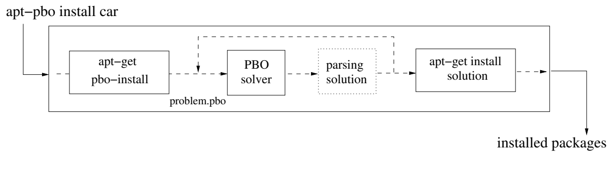

Figure 3: High level apt-pbo processing flow

The *apt-pbo* application is called with the operation *install* and the desired package as arguments, which map the usage of *apt-get*.

The components of the figure have the following role:

```
Require: Package to install $p_1, Pol$
 1: repeat
 2:    $(f, c) \leftarrow pboinstall(p_1, P_c, Pol, \mathcal{R}, PI)$
 3:    $S \leftarrow solver(f, c)$
 4:    $P_c \leftarrow \{\}$
 5:    for all $p_i \in S$ such that $p_i = 0$ do
 6:       $P_c \leftarrow check\_rdeps(p_i)$
 7:    end for
 8:    for all $p_j \in S$ such that $p_j = 1$ do
 9:       $P_c \leftarrow check\_rconfs(p_j)$
10:    end for
11: until $P_c = \{\}$
```

Figure 4: Iterative apt-pbo solving algorithm for package installation problem

- **apt-get pbo-install**: We have a modified version of *apt-get* installation software. Apt is one of the most used meta-installers and is adopted by different Linux distributions like Debian, Ubuntu and Caixa Mágica. Our modifications created a new method called *pbo-install* which, given a specific package, calculates the dependency tree and writes the PBO encoding. The PBO encoding is composed of PB-constraints and an objective function.

- **PBO solver**: The *problem.pbo* formula is solved by the PBO solver. We have used and tested different solvers to be introduced in section 4.4.

- **parsing solution**: apt-pbo has a module that parses the solver solution and, if necessary, establishes a new iteration with *apt-get pbo-install.*

- **apt-get install solution**: When the final package set solution is reached, the user is asked for permission and the removal and installation of the packages are performed using apt and dpkg / rpm.

The apt-pbo algorithm in Figure 4 describes the proposed approach to the process of finding the solution taking into account the available set of packages and the user defined policy regarding installation. The solution is the set of packages that change status due to the install transaction. A change of status occur when a package is marked to be removed or to be installed.

Given a package to install p_1, the universe of available packages $\mathcal{R}$ (repositories), the installed set of packages PI and the user policy regarding installation Pol we start by encoding the problem as PBO calling *pboinstall* . This procedure returns f as the cost function that minimizes the problem having the user policy Pol and returns c as the pseudo-Boolean constraints that must be obeyed during the optimization process.

Constraints c represent a directed acyclic graph that aggregates all the packages upon which p_1 depends on or conflicts with as well as their own dependencies and conflicts.

The installation problem encoded in PBO is passed to the *solver* that returns the best solutions (line 3).

After having a possible solution returned by the solver, we will check all elements in S that have a reverse relation outside c. Recall that S only contains packages that changed status. If package p_i is proposed to be removed, we will check for p_i reverse dependencies[4] and add them to p_c (line 6). The same is done to P_j, a new package that will be installed, but with its reverse conflicts.

We continue until the solution does not hold any reverse dependency not yet added to P_c.

In the next sections, we will discuss criteria that will be used to define the objective function and present our proposals.

3.2 Optimization of the Criteria

Although different sets of dependency packages can be found for installing a specific package, it is possible to rank solutions as more or less desirable. The best solution may not be the same for every user, as it depends on how the packages wil be use, the nature of the system (development, production, home PC, critical data server,...) and the limitations of the resources.

User preferences are ultimately encoded as PBO although they can be expressed initially in formats like CUDF (Common Upgradeability Description Format) [25] and then encoded into PBO.

A number of policies can be used to optimize the resulting installation [27]. Three of them were implemented in *apt-pbo*:

A **Removal policy** - Number of installed packages that will be removed (less is better).

B **Number policy** - Number of total packages that will be installed (less is better).

C **Freshness policy** - Number of packages in the system that are not the newer ones (less is better).

Other policies could be easily considered, such as the total download size, the final install size or the popularity. From our empirical experience with users in the scope of the deployment of a Linux distribution present in more than 650,000 computers, we found the criteria A, B and C as preferable.

Although heuristics can and have been used in the past to optimize a single criterion [18], that approach does not allow hybrid solutions or flexible changes on the relative values of each factor in run-time.

Optimization of only one criterion leads in most cases to a loss of quality with respect to other criteria. This happens since some of them are correlated and have conflicting objectives. For instance, getting a fresher package will typically lead to a larger one, since new features are introduced and size tends to increase. This is also true for *removal* and *freshness*. If we want to avoid removing packages, this will most probably compromise the freshness approach.

In the next section we will formulate the installation problem as a pseudo-Boolean optimization problem. The goal is to obtain the solution that best fits the user preferences expressed as multi-dimensional criteria.

car2 $\geq$ 1
engine2 + engine1 - car2 $\geq$ 0
engine2 + engine1 $\leq$ 1
wheel2 + wheel3 - car2 $\geq$ 0
wheel2 + wheel3 $\leq$ 1
door1 + door2 - car2 $\geq$ 0
door1 + door2 $\leq$ 1
turbo1 - engine1 $\geq$ 0
tyre1 + tyre2 - wheel3 $\geq$ 0
tyre1 + tyre2 $\leq$ 1
window0 + window1 + window2 - door2 $\geq$ 0
window0 + window1 + window2 $\leq$ 1
glass1 - window1 $\geq$ 0
glass2 - window2 $\geq$ 0
glass2 + tyre2 $\leq$ 1
glass1 + glass2 $\leq$ 1

Figure 5: Car problem constraints definition

3.2.1 PBO Encoding for Dependency Solving

The PBO encoding is done in two steps. First, the constraints are defined using the dependency tree graph. Second, the objective function is defined by weighting different criteria.

Constraints Definition

In a pseudo-Boolean formula, variables have Boolean domains and constraints are linear inequalities with integer coefficients.

Encoding relations of the dependency tree as constraints is a straightforward task. The following translations will be used:

- *Installation*: p_1 is the package that we want to install: $p_1 \geq 1$. In the context of PB constraints, saying variable p_1 has to be assigned value 1 implies having a "greater or equal" constraint.
- *Dependency*: p_1 depends on x_1 should be represented as $x_1 - p_1 \geq 0$. This means that installing p_1 implies installing x_1 as well, although x_1 may be installed without p_1. If p_1 also depends of y_1, we should add $y_1 - p_1 \geq 0$.
- *Multiple versions*: If a package p_1 requires the installation of a package x having different versions, for example x_1 and x_2, then we should encode the requirement that installing package p_1 requires installing either package x_1 or package x_2. Hence, such requirement may be encoded with constraint $x_1+x_2-p_1 \geq 0$.
- *Conflicts*: If a package has an explicit conflict with other package, for instance if y_3 conflicts with x_1, then this conflict is encoded as $x_1+y_3 \leq 1$. Remember that each pair of different versions of the same package is considered a conflict.

For the example given in figure 2, the constraints would be defined as presented in figure 5.

Objective Function Definition

The quality of the solution relies on the definition of the objective function. The policies proposed in section 3.2 can be easily translated into PBO encoding.

Minimizing Package Removal

To minimize the number of removed packages, even if newer packages exist, one should use the following *objective function*, where $PI'_1..PI'_N$ are the packages already installed:

[4] A dependency of p_1 is a package p_d that must be installed in order to install p_1. A p_1 reverse dependency is a package p_{rd} that depends on p_1.

$$f_1(P) = \min(1 - PI'_1) + ... + (1 - PI'_N)$$

In order to minimize the objective function, the solver will try to set variable PI_i to 1 which will imply not removing installed packages.

Minimizing the Number of Installed Packages

In this case, the total number of packages installed in the system is to be minimized. Having $P1..PN$ as the new packages targeted to be installed - either existent or new - the objective function will be:

$$f_2(P) = min\ P1\ +\ ...\ +\ PN$$

Maximizing the Freshness of Packages

If the user wants the most recent version of the packages, then the objective function should make the system install the most recent versions, independently of what is already installed.

Consider $P1_1..P1_{k1}$ to be different package versions or releases of package $P1$. Also, consider $v(P1_1)$ to be the normalized distance (a constant, for the purposes of the PBO problem) between the package $P1_1$ and the newest version present in repository R. Then the optimization function is:

$$\begin{aligned} f_3(P) =& min\ (P1_1 * v(P1_1) + ... + P1_{K1} * v(P1_{K1})) + \\ & (P2_1 * v(P2_1) + ... + P2_{KN} * v(P2_{KN})) + ... \end{aligned}$$

The value of $v(Pi_{Ki})$ is zero if the package is the newest in the repository.

Optimization of only one criterion leads in most cases to a loss of quality with respect to other criteria. This happens since some of the criteria are correlated. For instance, getting a fresher package will typically lead to a larger one, since new features are introduced and size tends to increase. This is also true for *removal* and *freshness*. If we want to avoid removing packages this will most probably compromise the freshness approach. In the next section we propose an approach to overcome this problem.

3.3 Multicriteria Optimization

Trying to satisfy different criteria when finding the set of packages for a software installation falls in the multicriteria decision making (MCDM) set of problems [8] and there is previous work on applying PBO to this research area [1].

Apt-pbo integrates the different objective functions of the previous section as a multiobjective problem (MOP), seeking the efficient solutions in the sense of Pareto optimality [9]:

$$min\ (f_1(P), f_2(P), f_3(P))$$

with P as the available packages and f_1, f_2 and f_3 as the existent objective functions.

The multiobjective problem is solved transforming it into a single objective problem through *weighted sum scalarization* of the form:

$$min\ \sum_{k=1}^{3} \lambda_k \cdot f_k(P)$$

where we minimize an additive function of the scalar product of λ that denotes the user-defined coefficients and $f_k(P)$, i.e. the objective functions defined before.

Apt-pbo uses the following coefficients, λ, representing the overall utility for the user:

- Removal Cost (W_r): weight given to the cost of a removal of a package. This coefficient is typically large when applied to a server in a critical environment.
- Presence Cost (W_p): weight given to the presence of a new or an already installed package in the solution.
- Version Cost (W_v): weight representing the cost of having an older version in the solution when a newer one exists.

The objective function is then defined as:

$$min\ (W_r \cdot f_1(P) + W_p \cdot f_2(P) + W_v \cdot f_3(P))$$

Instead of a weighted sum scalarization transformation, we could easily compare the criterion vectors ($f_1(P), f_2(P)$, $f_3(P)$) in lexicographic order but then would loose some of the flexibility introduced by the multiobjective approach.

4. EXPERIMENTAL RESULTS

We performed experiments on a large set of different repositories, packages and systems hosted at O2H Lab cluster of 184 Xeon CPU cores[5] with Linux installed in Xen virtual system machines and inside a *chroot* environment. In the next sections we report the results of this evaluation.

4.1 Experimental Methodology

The goal of the experiments performed was to simulate the installation of software in a Linux environment and test the various criteria under different environments.

We performed two groups of tests: one more oriented towards the finding of a solution and another more focused on the characterization of the solution itself.

The tested applications were:

- *apt-get*: the most used meta-installer among Linux distributions. *Apt-get* was called with *–no-install-recommends* to assure that the goal was exactly the same for all tested applications.
- *apt-pbo*: the tool proposed in this paper and tested with three different policies(freshness, removal and number). Each policy has a multicriteria function defined and is given a higher weight to the corresponding focus.
- *Smart*: a meta-installer developed in Python with heuristics to solve complex cases [19].
- *aptitude*: meta-installer with its own engine for dependencies solving [5].

4.2 Experimental Data

The data was generated using Linux Debian Lenny release as the operating system. Lenny is considered to be a stable version of Debian and is widely used. This Lenny installation has the following characteristics:

[5] The infra-structure is integrated in the ADETTI / ISCTE centre of RNG Grid.

- The base system is a Linux Debian *Lenny* system with Lenny official, updates, backports and debian-multimedia repositories active, which account for 26,251 distinct versions of packages, 23,760 normal packages and 165,360 dependencies.

As a common basis for all the tests, we use a typical Linux installation without X11 graphical interface and comprising 338 base packages. Some of the installed packages are from Debian Sid, the development version. All tests were performed using a batery of 1,000 installation transactions of the most popular packages. This information was provided by the Debian popularity contest [21]. Using the popularity ranking contributes to assure that the tests are representative. For this purpose a scheduler / job manager was developed in Perl which was responsible for scheduling the installation jobs in the cluster, process the outputs, store the logs with the results of the installation and control the timeouts[6].

4.3 Comparing Meta-Installer Performance

The first phase of benchmarks consisted in analysing the main existent Linux meta-installers (*apt-get*, *smart* and *aptitude*) against *apt-pbo*. The time measured refers to the time between the call of the application and the return of the solution. Retrieval of packages and installation was not measured because network latency might introduce spurious noise. The time spending on building the cache (*apt-get update / smart update*) was not taken into account since it is performed in the same way by all meta-installers.

Table 1: Benchmarking of meta-installers (results in seconds)

Meta-installer	Average Time	Standard Deviation
Apt-get	00:00.21	00:00.17
Aptitude	00:00.62	00:00.15
Smart	00:02.63	00:00.25
Apt-pbo	00:03.77	00:01.91

The results presented in table 1 reflect the comparison of *apt-pbo* and other used meta-installers. In this test, for *apt-pbo* was chosen "removal" user-policy as it is eventually the default of a normal user and it is the one which is closer to other meta-installers solution.

Although *apt-pbo* takes longer than the other tools, it still provides a solution in a reasonable time (3.77 seconds, on average) that does not compromise the user's experience of a package installation.

Figure 6 presents the time taken by each meta-installer to solve a package installation problem varying the size of the problem (measured in terms of installed packages). As expected, with *apt-pbo* the time of the execution grows with the size of the problem but the increasing is progressive. This fact is replicated with the standard deviation results depicted in table 1, which shows a low standard deviation.

Apt-pbo allows the use of different user policies for the selection of the packages to install (section 3.2.1). Each user policy has different weights mapping the importance of number, removal or 'freshness' of packages.

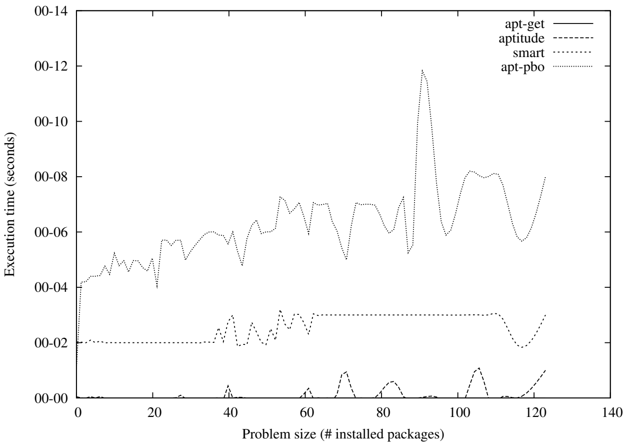

Figure 6: Execution time of meta-installers

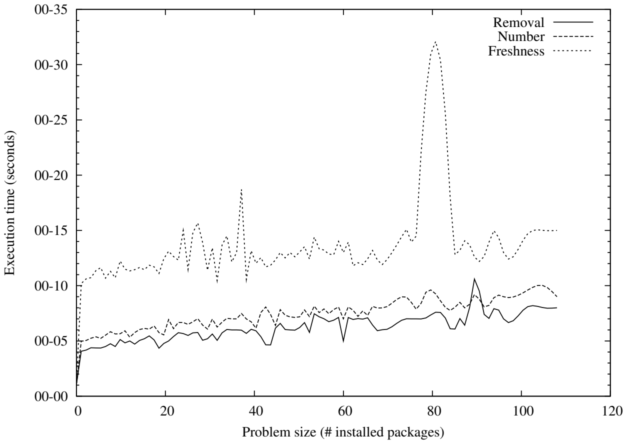

Figure 7: Execution time of *apt-pbo* according to different user policies

In figure 7 and table 2, we analyse the *apt-pbo* execution time for different size problems when changing the user-policy (removal, number and freshness). The freshness policy involves the installation of a large number of packages which requires more time not only in the solving part but also in parsing the solution.

Table 2: Benchmarking of apt-pbo user policies (results in seconds)

Meta-installer	Average Time	Standard Deviation
Removal	00:03.77	00:01.91
Number	00:04.26	00:02.24
Freshness	00:08.30	00:05.13

4.4 Comparing Different PBO Solvers

A comparison of SAT solvers has been done intensively in the past through international competitions and benchmarks [3, 17].

Since the solving algorithm can benefit greatly from the structure of the problem, it was considered important to test different PBO solvers in this problem.

[6] The time limit was set to 300 seconds. After 300 seconds the job was killed.

As mentioned in section 3.1, *apt-pbo* is structured in modular form, allowing the replacement of one solver instance by another compatible solver.

For testing purposes, four solvers were considered:

- *minisat+* [7]: based on minisat, a well known SAT solver, minisat+ encodes PB-contraints into SAT.
- *bsolo* [10]: bsolo is a PBO solver which was first designed to solve instances of the Unate and Binate Covering Problems (UCP/BCP) and later updated with Pseudo-Boolean constraints support.
- *wbo* [16]: participated in PB'09 competition, this solver uses Weighted Boolean Optimization which aggregates and extends PBO and MaxSAT.
- *opbdp* [2]: an implementation in C++ of an implicit enumeration algorithm for solving PBO.

Besides the solvers mentioned above, Pueblo [23] was also considered but not included since the only available version is dynamically linked and the libraries needed are old and not available in the testing infra-structure. Nevertheless, an old Linux system was installed (Debian Etch) and some adhoc tests were performed with Pueblo. These tests revealed that Pueblo has in general a bad performance for this specific type of problems and no further efforts to port Pueblo were made.

Table 3 summarizes the results. As can be verified, both *wbo* and *bsolo* are able to solve all the instances but *wbo* has a better performance (7.79 seconds on average per transaction). *Minisat+* comes in third place, not only with a low number of instances solved, 355, but also with a poorer performance, taking on average more than two minutes to solve a problem.

Table 3: PBO solvers benchmarking

	wbo	bsolo	minisat+	opbdp
# Solved	1,000	1,000	355	47
# Timeouts	0	0	645	953
Average time	00:07.79	00:04.45	02:30.16	07:16.49

Figure 8 compares *wbo* and *bsolo* varying the number of the installed packages. There is a smooth growth by *wbo* and a more unstable line of growth in a much more unpredictabile fashion by *bsolo*.

4.5 Unmet Dependencies Analysis

The unmet dependencies are reported by Apt and other meta-installers when a required dependency is not found. This could be either the result of a broken dependency in the repository or the solver just being unable to find it in a proper way. The first topic has been investigated by different groups [14, 31] and has registered a significant progress. In this section we present the results that demonstrate the inability of today's meta-installers - namely *apt-get* - to find a solution in every case and, on the other hand, the encouraging results of *apt-pbo*.

A specific example of a unmet dependency is presented in figure 9.

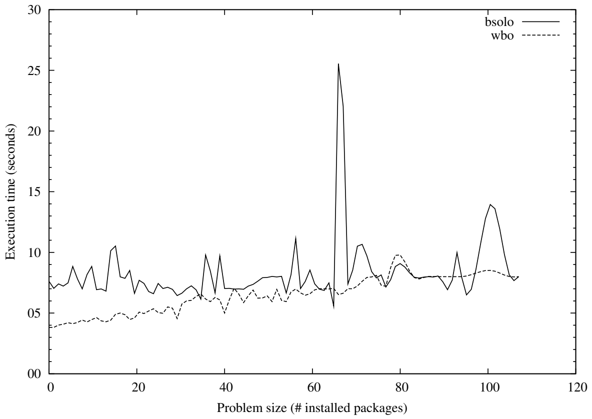

Figure 8: PBO solvers graph

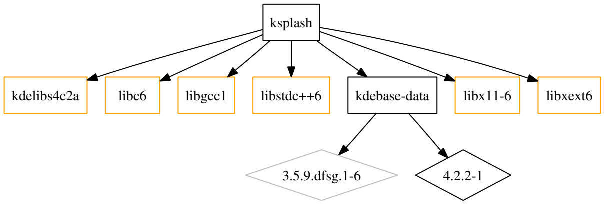

Figure 9: Example of an unmet dependency - ksplash package.

In figure 9, the package *ksplash* is represented with multiple dependencies. One of them is *kdebase-data* that is available under two versions: *4.2.2-1* and *3.5.9.dfsg.1-6*. The *ksplash* package has two constraints that say that it requires *kdebase-data* (<< 4:3.5.9.dfsg.2) and *kdebase-data* (>> 4:3.5.9.dfsg.1) which match with existent package *3.5.9.dfsg.1-6.*

Apt-get is only able to backtrack into the newest version and unable to find the solution. This is the simplest category of problems, but over a large group of relations presents a hard problem. Recall that package dependencies can have version comparison operators $(=,\geq,>,<,\leq)$ as well as Boolean operators (*depends on debconf OR debconf-2.0*).

The second group of unmet dependencies problems are related with conflicts. A package might conflict with another package which results in the need to install another version.

For the assessment of unmet dependencies extension, we performed tests over a universe of the top 1,000 packages of *popcon* list. *Apt-get* reported the inability to install 123 packages due to unmet dependencies. These packages, correspond to 12.3% of the total packages not being available to the user, which is an unusual high rate. This high rate is justified by a base system that contains a mixture of packages from both Lenny and Sid.

Table 4: Metainstallers and solution characterization

Meta-installer	Possible Solutions	No Solutions	Wrong Solutions
Apt-get	591	123	0
Aptitude	713	0	1
Smart	714	0	0
Apt-pbo	714	0	0

Apt-pbo, as well as *Smart*, was able to find a solution to every case of the 1,000 packages. *Aptitude* performed also well with the exception of a false positive of a package that was not installable and was reported as so.

4.6 Package Installation Solutions Assessment

The finding of a solution by *apt-pbo* is important only if it is as good or even better than solutions returned by existent tools and techniques. In what follows we show that the *apt-pbo* solutions are equivalent and sometimes better: fewer packages, newer packages installed or few packages removed. This varies according to the policy defined.

Table 5 refers to tests with Lenny as characterized before. The tests generated the transaction (install, upgrade,...) of 83,545 packages in the defined environment. In each column is presented the sum of packages for the 1.000 installation transactions that changed status.

Table 5: Total package transactions by metainstaller (1,000 packages)

Tool	Installed	Updated	Removed	Downgraded
apt-get	7,766	17	12	0
aptitude	8,423	18	161	464
smart	7,786	92	124	479
PBO freshness	6,808	20,449	25	567
PBO removal	7,767	11	162	443
PBO number	7,729	13	102	500

As we can confirm analysing table 5, the results are very different between meta-installers. *Apt-pbo* with freshness policy proposes a lot more updates which is a critical point for users that want to have their system updated. This ability to update the system even further than the direct dependencies tree as also an impact on the *Removed* factor, being the lowest of all benchmarks. Apt-pbo removal will need to have its weighting tweaked to get better results than freshness.

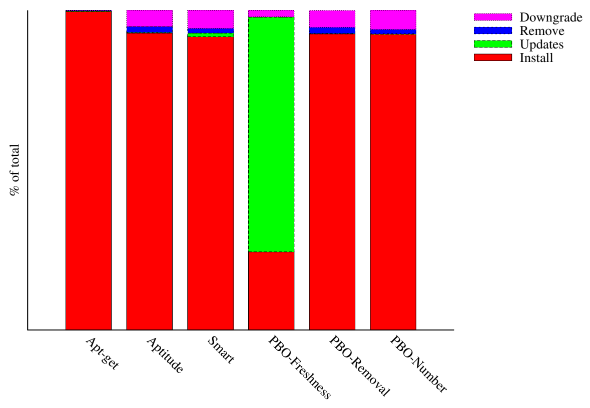

Figure 10: Relative transaction distribution by meta-installer

The graph in figure 10 provides a global overview of the data in table 5 and shows that the structures of the solutions are very different.

Table 6: Average package transactions for Apt policies

Tool	Installed	Update	Remove	Downgrade
PBO freshness	10.07	30.25	0.04	0.84
PBO removal	10.88	0.02	0.23	0.62
PBO number	10.82	0.02	0.15	0.7

In what concerns *apt-pbo* different policies, table 6 presents the average number of each transaction per installed package. *Apt-pbo* with the removal policy performed under expectations (0.23 seconds for the removal criterion on average per package installation). Hence, in the future the weights should be rebalanced to obtain better results in terms of removed packages. On the other hand, *apt-pbo* with number policy outperforms *Smart* and *Aptitude* with respect to *removal* factor compensating it with the downgrade which could be a good trade-off for critical servers.

The experimental results are very encouraging and *apt-pbo* is starting to be tested by real users. On the other hand, the PBO approach was able to find a solution to every installation problem, even when the original Apt installer was not capable of doing so. The solver finds rapidly a solution for the instances presented, and the package selection problem is not only manageable but also very efficient when using the PBO approach. The freshness cost function gets not only the solution with newer packages but also the minimal number of installed packages. The time consumed is very acceptable for a user interactive application.

4.7 Multi-criteria analysis

In section 3.3, we presented the theory behind the multi-criteria approach of *apt-pbo* and in the previous section we presented a pre-defined set of weights for different user profiles (freshness, removal and number). Focusing on *freshness* policy and fixing the other weights, we will now analyse the variation of version cost (W_v) with two values. In the first scenario, that we call conservative freshness, W_v is set to 3,000. In the second scenario, that we call aggressive freshness, W_v is set to 30..

Table 7: Varying weights in *freshness* policy (average)

	Aggressive	Conservative
Install	12.98	12.27
Updates	30.18	30.13
Remove	0.06	0.03
Downgrade	0.84	0.85
Time	00:11.18	00:11.50
Timeouts	0	22

From table 7, we conclude that although on average the changes are not so significant, they have important differences. For instance, the removal time doubles in the aggressive freshness because we are strongly interested in having updates, and so the compromise is to allow some packages to be removed. In the installation of 1,000 packages, 60 packages have been removed against just 30 packages removed in conservative freshness.

The best way to check differences is by analysing a real case: the installation of the *at-spi* package with different weights. *At-spi* is an assistant interface that provides accessibility to Gnome graphical environment and therefore exhibits several Gnome package dependencies.

Table 8: Installing at-spi package with conservative and aggressive freshness weights

	Aggressive	Conservative
Install	9	37
Updates	35	35
Remove	1	0
Downgrade	7	10
Time	00:19.45	00:17.37

The conservative freshness profile avoids the removal of packages as can be seen in table 8. However, the trade-off is extra packages for install and downgrade.

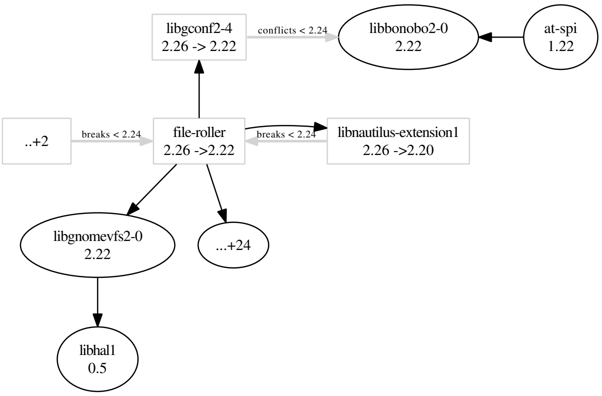

Figure 11: Package *at-spi* installation with conservative freshness

Figure 12 presents a graph of a sub-set of *at-spi* dependencies. In order to install *at-spi*, we need to install the package containing *libbonobo2-0* library. This package is only available in version 2.2 which conflicts with *libgconf2-4* package, thus resulting in the proposal of downgrade of this package from 2.26 to 2.22. However, package *file-roller* depends on the previous *libgconf2-4* version. In the conservative freshness, file-roller is downgraded causing the installation of 26 new packages and the downgrade of 3 packages.

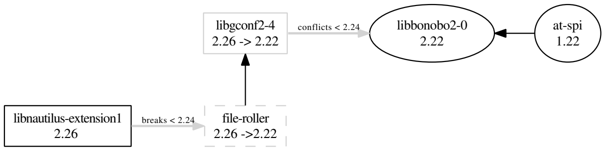

Figure 12: Package *at-spi* installation with aggressive freshness

The aggressive freshness (see figure 12) is more tolerant to removals. The removal of the *file-roller* package proposed in this profile avoids the massive installation of packages and downgrade of a few other. Having that file-roller is not a critical package but rather a graphical frontend for commandline compressing tools, this may be the best solution for a large group of users.

Observe that the simple lexicographical ordering of objective functions has this exact problem: in package installation realm we lose by independently optimizing a single criterion over all the others.

5. RELATED WORK

In the field of component-based systems, the use of PBO was proposed generically for the assembly of components in [15].

The use of Boolean Satisfiability (SAT) for solving the dependency problem has first been proposed in the context of the EDOS FP6 project [14] which had impact in other research efforts [12]. An alternative formulation using constraint programming techniques has been described in [18], including the use of different heuristics for improving the quality of the solution found.

Later on, the use of PBO has been independently proposed by two different research teams [27, 28]. The Opium tool [28] presents some drawbacks which in our opinion can invalidate the approach. First, it encodes the whole package universe as constraints which requires extra computation in opposition to a selective and incremental choice of the packages as presented in the algorithm of figure 4. Finally, it does not provide a formal description for the PBO encoding and the optimization function which is a critical part of a meta-installer proposal.

6. CONCLUSIONS

In this work we formulated the installability problem as a pseudo-Boolean optimization problem in order to be able to encode not only the constraints implied by the package dependencies but also the optimization criteria chosen by the user that characterize each installation.

An open implementation based on Apt has shown that the developed tool can be easily reconfigured to use this method, which can be effectively used for real installation problems.

After the intensive testing in Debian (DEB) and Caixa Magica (RPM) distributions, we propose to continue developing the current approach, to further extend the tool and to port it to other distributions. The enhancement of cost functions for the different criteria will be pursued with the goal of better performance and improvement of the solutions provided. Given the complexity of the problem, some techniques will have to be devised to abort the search process if, in some instances, it is taking too long to find a particular solution. However, we believe that such behavior will be more the exception than the rule, and that this method vastly improves the actual state of the art, where installers fail with relatively trivial problem sets and obtained solutions are far from perfect.

Acknowledgments

Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898. Paulo Trezentos thanks John Thomson for interesting discussions on this topic and proof-reading the article.

7. REFERENCES

[1] B. Alidaee, H. Wang, and Y. Xu. A pseudo-boolean optimization for multiple criteria decision making in complex systems. In *Proceedings of the 7th international conference on Computational Science (ICCS '07), Part IV*, pages 194–201, Berlin, Heidelberg, 2007. Springer.

[2] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical report, Technical Report MPI-I-95-2-003, Max Planck Institute., 1995.

[3] D. L. Berre, O. Roussel, and L. Simon. International SAT 2009 competition, 2009. http://www.satcompetition.org/2009/.

[4] L. Bixin. Managing dependencies in component-based systems based on matrix model. In *NETObjectDays'03*, 2003.

[5] D. Burrows. Modelling and resolving software dependencies, 2005. Debian.

[6] D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Towards maintainer script modernization in foss distributions. In *Proceedings of the 1st international workshop on Open component ecosystems (IWOCE '09)*, pages 11–20, New York, NY, USA, 2009. ACM.

[7] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, March 2006.

[8] M. Ehrgott. *Multicriteria optimization.* Lecture Notes in Economics and Mathematical Systems. Springer, 2000.

[9] M. Ehrgott and X. Gandibleux. A survey and annoted bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 2000.

[10] F. Heras, V. Manquinho, and J. Marques-Silva. On applying unit propagation-based lower bounds in pseudo-Boolean optimization. In *Proceedings of International FLAIRS Conference*, 2008.

[11] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. Technical report, Computer Science Department, University of Northern Iowa, 2004.

[12] D. Le Berre and A. Parrain. On SAT technologies for dependency management and beyond. *ASPL*, 2008.

[13] D. Le Berre and P. Rapicault. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems(IWOCE '09)*, pages 21–30, New York, NY, USA, 2009. ACM.

[14] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *International Conference on Automated Software Engineering (ASE'06)*, pages 199–208. IEEE Computer Society, 2006.

[15] P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07)*, pages 61–72, New York, NY, USA, 2007. ACM.

[16] V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted boolean optimization. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, 2009.

[17] V. Manquinho and O. Roussel. PBO competition 2009, 2009. http://www.cril.univ-artois.fr/PB09/.

[18] S. Mouthuy, L. Quesada, and G. Doom. Search heuristics and optimisations to solve package installability problems by constraint programming. Technical report, Department of CSE, UC Louvain, October 2006.

[19] G. Niemeyer. Smart package manager, 2009. http://labix.org/smart.

[20] Pietro Abate, Jaap Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *Proceedings of 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, 2009.

[21] D. project. Debian popularity contest, 2009. http://popcon.debian.org/.

[22] O. Roussel and V. M. Manquinho. Pseudo-Boolean and cardinality constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009.

[23] H. M. Sheini and K. A. Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:2006, 2006.

[24] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis in support of software maintenance. In *Proceedings of the third international workshop on Software architecture (ISAW '98)*, pages 129–132, New York, NY, USA, 1998. ACM.

[25] R. Treinen and S. Zacchiroli. Expressing advanced user preferences in component installation. In *Proceedings of the 1st international workshop on Open component ecosystems (IWOCE '09)*, pages 31–40, New York, NY, USA, 2009. ACM.

[26] P. Trezentos. Apt-pbo homepage, 2009. http://aptpbo.caixamagica.pt/.

[27] P. Trezentos, R. Di Cosmo, L. Lauriére, M. Morgado, J. Abecasis, F. Mancinelli, and A. Oliveira. New Generation of Linux Meta-installers. *Research Track of FOSDEM 2007*, 2007.

[28] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *29th International Conference on Software Engineering (ICSE 07)*, pages 178–188, 2007.

[29] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *Proceedings of the 17th IEEE international conference on Automated Software Engineering (ASE '02)*, page 241, Washington, DC, USA, 2002. IEEE Computer Society.

[30] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08)*, pages 63–74, New York, NY, USA, 2008. ACM.

[31] S. Zacchiroli, R. Di Cosmo, and P. Trezentos. Package upgrades in FOSS distributions: Details and challenges. In *First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*, October 2008.