Towards Efficient Optimization in Package Management Systems

Alexey Ignatiev
IST/INESC-ID, Lisbon, Portugal
aign@sat.inesc-id.pt

Mikoláš Janota
IST/INESC-ID, Lisbon, Portugal
mikolas@sat.inesc-id.pt

Joao Marques-Silva
University College Dublin, Ireland
IST/INESC-ID, Lisbon, Portugal
jpms@ucd.ie

ABSTRACT

Package management as a means of reuse of software artifacts has become extremely popular, most notably in Linux distributions. At the same time, successful package management brings about a number of computational challenges. Whenever a user requires a new package to be installed, a package manager not only installs the new package but it might also install other packages or uninstall some old ones in order to respect dependencies and conflicts of the packages. Coming up with a new configuration of packages is computationally challenging. It is in particular complex when we also wish to optimize for user preferences, such as that the resulting package configuration should not differ too much from the original one. A number of exact approaches for solving this problem have been proposed in recent years. These approaches, however, do not have guaranteed runtime due to the high computational complexity of the problem. This paper addresses this issue by devising a hybrid approach that integrates exact solving with approximate solving by invoking the approximate part whenever the solver is running out of time. Experimental evaluation shows that this approach enables returning high-quality package configurations with rapid response time.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Domain engineering; F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures; G.1.6 [**Optimization**]: Constrained optimization; K.6.3 [**Software Management**]: Software maintenance

General Terms

Algorithms, Performance, Management

Keywords

Package management, SAT solving, MaxSAT solving, Optimization, Minimal Correstion Subsets

1. INTRODUCTION

The complexity of *package management systems* stems from two main objectives: *personalization* and *reusability*. On the one hand, different users are expected to use their computers for different purposes and thus requiring different software artifacts on their computers but on the other hand, it is desirable to reuse software artifacts whenever possible. Nowadays package management systems, such as those behind *Linux distributions*, enable handling hundreds of thousands of software artifacts. This is possible due to organizing software into *packages* and *explicitly* recording relations between those packages. Once a user requires for a package to be installed, a program called the *package manager* consults the *package repository* and determines the package's *dependencies*, i.e. which other packages also need to be installed. Or, it might determine that some other package *conflicts* with the one that is supposed to be installed.

Due to such relations between packages, maintaining a particular package configuration consistent poses a computational challenge. Indeed, to determine whether a package can be installed into a given configuration of packages is NP-complete [1]. Consequently, current package management tools are known to be *not* complete, i.e. there are situations when they are not able to find a solution even if it exists [2].

All of the above motivates the formulation of *package upgradability problem* (or *package installability*). The input to the problem is the *initial configuration*, *relations between packages* (conflicts and dependencies), and *user's request* [3]. The output to the problem is a configuration of packages that satisfies the relations between them and also the user's request. For the purpose of this article we consider a request to be a set of packages to be installed and a set of packages to be removed. In our implementation, however, the upgrade request is also supported; such can be modeled as removal of an old version and installation of a new version.

In practice, users are not likely to be satisfied with an arbitrary solution to the package upgradability problem. For instance, users typically expect that the resulting configuration of packages will not differ too much from the initial configuration. Or, the user may wish to bring all the packages up to date. For this purpose, we also enable the user to specify *optimization criteria* that are to be optimized in the resulting package configuration.

While optimization criteria are indispensable for practical application of package management, they make the problem computationally more complex. However, number of approaches were developed in the recent years relying on various technologies, such as *Answer set programming*, *Pseudo-Boolean optimization*, or *MaxSAT* [1, 2, 4, 5]. It has been demonstrated that these approaches are mature enough to be applied in practice. Most notably, the development environment Eclipse IDE [6] employs the pseudo-Boolean-based engine P2 to drive package management.

Dedicated optimization solvers have an important advantage over heuristic-based algorithms and that is, they guarantee to find the optimal solution given sufficient amount of time. The runtime, however, might be an issue from a practical perspective due to the high computational complexity of the problem. It is unacceptable for a user to wait hundreds of seconds. Thus, for successful integration of the tools into practice, they must be able to find a solution within a number of seconds [7]. Therefore, an optimization solver may be forced to stop in the middle of the computation and return some solution, which is possibly non-optimal. This is indeed the case in the solver P2, which is used in Eclipse IDE. Observe that Eclipse IDE is used by literally millions of users, and so millions of users use P2 on a regular basis.

To our best knowledge, there is no approach to the package upgradability problem that would be dedicated to finding good approximate solutions within practical time limits. Instead, the current approaches respond with a "best so far" solution, which provides little or no guarantees.

The discussion above can be summarized into two main observations:

1. Complete optimization solvers *can* be successfully used in practice.
2. Users require *rapid* responses and therefore it is likely that the solver will run out of time.

These observations lead to the following challenge: *if a complete package upgradability solver is running out of time, it must be able to produce a* good *solution nevertheless.*

In response to this challenge, this paper proposes an approach for finding good near-optimal solutions to the package upgradability system. At the same time, it enables construction of a *hybrid solver*. This hybrid solver integrates a complete approach with the approximation approach in a way that the approximation approach is invoked only when needed. Thus, it is providing us with the best of the two worlds: completeness and rapid response.

We have implemented this hybrid solver on top of an existing tool for the package upgradability problem. We have performed an extensive experimental evaluation, where our hybrid approach was compared to the tool P2. This evaluation demonstrates that the hybrid approach enables resolving much larger number of instances than P2 within a rapid timeout. Further, the provided responses are significantly closer to the optimum than the responses of P2.

This article is organized as follows. Section 2 introduces concepts and notation used throughout the paper. Section 3 describes the proposed approach for solving package upgradability; this section is the main contribution of the paper. Section 4 presents an experimental evaluation of the proposed approach. Related work is summarized in Section 5 and finally the paper concludes by Section 6.

2. PRELIMINARIES

A number of formalisms exists to explicitly capture dependencies between software artifacts. Prominent examples are *feature models*, which capture dependencies of software

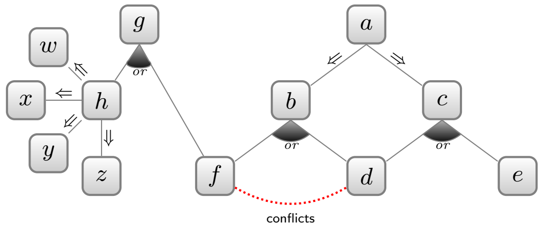


Figure 1: Example dependencies

components of a software system including variability [8]. Linux distributions rely on *package management systems* where software components are organized into packages captured in standardized formats such as *RPM* [9] or *DEB* [10].

A *package manager* is used to maintain a particular configuration of the system in a consistent state, i.e. in a state where dependencies between packages are not violated. Such package manager must be able to receive requests from the user to install or uninstall certain packages. Whenever such request is obtained, the package manager must compute a new configuration satisfying both the request and the package dependencies.

Figure 1 schematically shows a small example of dependencies between packages (the diagram uses primitives similar to the ones used in feature modeling [8]). Imagine now that we wish to install package a and at the same time we wish to minimize the number of installed packages. Since a requires both b and c, these have to be installed as well. Now the packages f, d, and e can be used to fulfill dependencies of b and c. In this case, as we want to minimize the number of installed packages, we install d as that satisfies dependencies of both b and c. Now let us consider another scenario where we wish to install both a and g and again, we wish to minimize the number of installed packages. If we solve the problem greedily and install a, b, c, and d, we will end up with a suboptimal solution because package d excludes the use of package f and therefore packages h, x, y, z, w have to be used to satisfy dependencies of package g.

In this paper we consider packages described in the language *Common Upgradability Description Format (CUDF, e.g. see [11])*. CUDF was developed by the scientific community and thus is more amenable to scientific analysis and is supported by a number of freely available tools [12]. At the same time, CUDF is a realistic language which was successfully integrated into existing tool-chains [13].

For the sake of succinctness and clarity, the presentation in this paper does not consider the full extent of the CUDF language (but the language was considered in full for the experimental analysis). Every package has a *name* and a *version*, where a name is a string and version is a positive natural number. A package may *depend* on some other packages or it may *conflict* with some other packages. A package can also be attributed its *size* or the set of packages that are *recommended* to be installed with it. The attribute *installed* tells us if the package is installed in the current package configuration. An example of a package in CUDF follows.

```
package: openssl
version: 5
depends: libc6 >= 9, libssl1.0.0 >= 5,
         zlib1g >= 14
conflicts: openssl
installed: true
installedsize: 1184
```

The package's name is `openssl` in its version is `5`. For its functioning the package requires certain versions of `libc6`, `libssl1.0.0`, and `zlib1g`. For instance, if this package is to be installed, there must be also installed a package `libc6` with the version greater or equal to `9`[1]. The `conflicts` specification in this case says that the package conflicts with all packages with the same name except with itself.

A solver for the package upgradability problem is given a set of packages, each described as the one in the example, and it outputs a new package configuration as a set of packages that are installed in it.

2.1 Package Upgradability Problem

The problem that a package manager has to solve whenever it faces a user request is called the *package upgradability problem.* A package manager operates on a *package universe* (or *repository*), i.e. the set of packages that are visible to it. A *user request* comprises packages that the user wishes to be *installed* and the set of packages that the user wishes to be *uninstalled.* Additionally, the package manager may be given a criterion according to which it should optimize when looking for the new configuration. Such criterion can be for instance minimizing the number of newly installed packages or minimizing the number of packages that were installed in the initial setup but are uninstalled in the solution.

More formally, a package manager is given a package universe U and a user request represented as a pair (I, R), where I represents the packages that are to be installed and R are the packages to be removed. The output is a set of packages N such that N contains all of the packages from I; does not contain any of the packages from R; and all the package dependencies and conflicts are respected in N. If such N does not exist, the package manager reports a failure.

Optimization criteria are a complex issue and it is beyond the scope of this paper to describe them in full detail. We consider optimization criteria used in the MISC competition [12]. These criteria typically rely on some additional attributes of a package (e.g. see `openssl` above). Most notably, a package can have the attribute `installed`, which signifies that the package is currently installed in the system (if the attribute is not present, it defaults to `false`). This attribute enables specifying criteria that express that the new installation should differ from the original one as little as possible. Multiple sub-criteria are combined by a *lexicographic ordering.* For instance a criterion specified as `(-count(removed), -count(changed))` defines a lexicographic order where we wish to minimize the number of removed and the number of changed packages, and, minimizing the number of removed packages has a higher priority than minimizing the number of changed packages.

2.2 Boolean Satisfiability and Maximum Satisfiability

To decide whether a package upgradability problem has a solution or not, is NP-complete [1]. Hence, appropriate tools for solving the problem need to be chosen. The package upgradability problem naturally translates to *Boolean satisfiability (SAT)* [2], *constraint satisfaction (CSP)* [3, 4], or *answer set programming (ASP)* [5] paradigms. In this paper we consider the SAT paradigm and for the optimization part of the problem we consider a *maximal satisfiability (MaxSAT)* [14] formulation. For such the following standard definitions are considered.

[1] The version numbers of the packages does not necessarily have to correspond to the version of the software component that the package encapsulates.

A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of finitely many literals (possibly none). A formula is in *conjunctive normal form* (CNF) if it is a conjunction of finitely many clauses (possibly none). As is common, whenever convenient, a clause is treated as a set of literals and a CNF as a set of clauses. An *assignment* to a set of variables X is a function from X to the constants 1 and 0, corresponding to *true* and *false*, respectively. An assignment *satisfies* a formula ϕ iff the formula evaluates to 1 under that assignment. The SAT problem is the problem of deciding whether a CNF ϕ has a satisfying assignment.

Given a CNF ϕ, the *MaxSAT problem* consists in finding an assignment to the variables of ϕ that maximizes the number of satisfied clauses (clauses that evaluates to 1 under the assignment).

The *partial MaxSAT problem* is given as an input two CNF formulas: the *hard clauses* ϕ_H and the *soft clauses* ϕ_S. A solution to such problem is an assignment that satisfies all the clauses in ϕ_H and maximizes the number of satisfied clauses in ϕ_S. For instance, for the CNFs $\phi_H = \{(x \vee y)\}$ and $\phi_S = \{(\neg x), (\neg y)\}$ the assignment ν defined as $\nu(x) = 1, \nu(y) = 0$ is a solution satisfying one of the soft clauses. In contrast, the assignment μ defined as $\mu(x) = 1, \mu(y) = 1$ is not a solution since it does not satisfy any of the soft clauses.

MaxSAT also has a weighted variant, which is useful for optimization with weights. A *partial weighted MaxSAT* consists of hard clauses ϕ_H and weighted soft clauses ϕ_S. A solution to a partial weighted MaxSAT is an assignment that satisfies all of the hard clauses and maximizes the number of weights of soft clauses that are satisfied. We will denote a weighted clause C with weight W as $(W\,;C)$. For example, if hard clauses are defined as $\phi_H = \{(x \vee y)\}$ and soft clauses as $\phi_S = \{(20\,; \neg x), (10\,; \neg y)\}$, the problem has a single solution which is the assignment $\nu(x) = 0, \nu(y) = 1$. Intuitively, the hard clauses prescribe that one of the x and y must be 1. At the same time, however, the soft clauses force both of the variables to 0. Consequently, we must pick one of the variables to be 1 and one of them to be 0 but because the clause $(\neg x)$ has a larger weight, x is set to 0.

3. APPROACH

In our approach to the package upgradability problem, we follow the popular techniques for translating dependencies between packages into Boolean logic. Then, we use (weighted) MaxSAT to express optimization criteria.

First we show how to decide whether a package upgradability problem has a solution or not can be encoded as a SAT problem. Each package p is assigned a unique Boolean variable v_p, representing whether the package is installed in the solution being constructed. The clauses capturing the dependencies and conflicts are generated according to the following rules. If a package p requires one of the packages $r_1, \ldots, r_n$, generate the clause $(\neg v_p \vee \bigvee_{i \in 1..n} v_{r_i})$. If a package p conflicts with a package o, generate the clause $(\neg v_p \vee \neg v_o)$. To encode the user request, we follow the following two rules. If a package p is requested to be installed, generate the clause (v_p) and when it requested to be uninstalled, generate the clause $(\neg v_p)$.

Once a CNF formula is constructed as described above, we can invoke a SAT solver to decide whether the considered upgradability problem has a solution or not. If it does, a SAT solver also provides us with a satisfying assignment to the constructed CNF. This assignment tells us which packages should be installed. In particular, a package p is installed if and only if the variable v_p is assigned to the value 1.

As noted above, from the user point of view, it is not sufficient to just find any solution to the package upgradability problem but we want to find a solution that is optimal with respect to some criteria. We consider the encoding of the optimization problem as (weighted) MaxSAT. For such, all the clauses above are considered to be the hard clauses — these must be satisfied by any solution. Soft clauses are used to encode the criterion to be optimized.

For succinctness reasons, we will not describe all considered optimization criteria and their encodings; the reader is referred to existing literature for further details [15]. For illustration let us consider the criterion where we wish to minimize the number of *removed packages*. A package is considered removed if there is some version of the package in the initial package configuration and there is no version of that package in the final configuration. For such we first collect the set of package names $\mathcal{I}$ for which there is some package in the initial configuration. Then for each of these names $n \in \mathcal{I}$ we construct the soft clause $(\bigvee_{p.\mathtt{name}=n} v_p)$. Intuitively, this soft clause enforces that at least one version of the package is installed. Recall that in MaxSAT, soft clauses are not necessarily satisfied. However, a solution to a MaxSAT guarantees that the largest possible number of soft clauses is satisfied. Consequently, in this particular case, the smallest possible number of packages will be removed.

To show how weighted MaxSAT is useful, consider the criterion where we wish to minimize the total size of the resulting package configuration. For such, for each package p generate the soft clause $(p.\mathtt{size}\,;\,\neg v_p)$. Such clause tries to enforce that p is *not* installed. However, more emphasis is put on larger packages. Observe that if all packages have the same weight, the problem is the same as minimizing the total *number* of installed packages and then unweighted MaxSAT is sufficient.

To illustrate the above, consider the right-hand side of Figure 1 under the assumption that we wish to install package a and minimize the total number of installed packages. The hard clauses are $(\neg v_a \vee v_b)$, $(\neg v_a \vee v_c)$, $(\neg v_b \vee v_f \vee v_d)$, $(\neg v_b \vee v_d \vee v_e)$, $(\neg v_f \vee \neg v_d)$, and (v_a). Note that the last clause represents the user request to install package a. The soft clauses are $(\neg v_a)$, $(\neg v_b)$, $(\neg v_c)$, $(\neg v_d)$, $(\neg v_f)$, and $(\neg v_e)$. Observe that the soft clause $(\neg v_a)$ can never be satisfied in a solution because of the hard clause (v_a).

3.1 Boolean Lexicographic Optimization

So far we have discussed how to translate a single optimization criterion into MaxSAT. In practice, however, *multiple* criteria might be required. For instance, the user may wish to find a package configuration that differs the least from the existing configuration but at the same time it does not occupy too much space on the disc. A natural way how to combine multiple criteria is via the *lexicographic ordering* where the individual criteria are given different priorities. So for instance if we have two criteria C_1 and C_2 where C_1 is given the higher priority, a total optimum must first optimize C_1 and only then C_2.

Since MaxSAT does not have native support for criteria organized in a lexicographic order, we need a way to communicate such order to the solver. A lexicographic order can be encoded as an optimization of a single function by given appropriate weights to the individual criteria [4,16]. In practice, however, it has been observed that it is significantly more efficient to optimize for each of the criteria *individually* [17]. In particular, a solver begins by optimizing the most important criterion, and fixes the weight of the optimum once it has found it. Then, it proceeds with optimizing the second most important criterion, etc. This technique is referred to as *Boolean Lexicographic Optimization* (*BLO*).

More formally, package upgradability problem considered as Boolean lexicographic optimization problem is represented as follows:

$$\begin{array}{ll} \text{lexmax} & (f_1, f_2, \ldots, f_k) \\ \text{s. t.} & \phi_H \end{array} \tag{1}$$

where, f_i, $1 \le i \le k$ denote the objective functions to optimize and ϕ_H denotes the hard constraints. The function f_1 has the highest priority, the function f_2 has the second highest priority, etc.

Recent work on package management problems [17] proposes solving (1) iteratively as follows:

$$\begin{array}{ll} \max & f_i \\ \text{s. t.} & \phi_H \\ & f_j = M_j, j = 1, \ldots, i-1 \end{array} \tag{2}$$

where M_j is the optimum value for the cost function f_j, given that preceding functions have already been optimized.

In recent years, different approaches for implementing (2) have been proposed. A special place is occupied by methods based on SAT — namely, MaxSAT [4, 15, 17], Pseudo-Boolean optimization [4, 17, 18], ASP [5, 19], etc. Among these, the use of MaxSAT has been shown competitive with alternative approaches. There have been developed a number of MaxSAT-based package upgradability solvers, e.g. inescp [20], PackUp [15].

The use of MaxSAT entails encoding each f_i as a set of soft constraints ϕ_S^i; this is done as described in the previous section. At each iteration i it must be guaranteed that optimal values of the functions f_j, $j < i$, which were already found, are fixed. Fixing the value of f_j is done by introducing additional hard constraints $\mathtt{CNF}(f_j = M_j)$, where $\mathtt{CNF}$ is a function that constructs a CNF encoding of the constraint $f_j = M_j$. There are several ways to implement the function $\mathtt{CNF}(f_j = M_j)$, but their description is out of the scope of this paper. The reader is referred to [15, 17] for further details. At each iteration, the MaxSAT problem to be solved is characterized by a set of soft constraints ϕ_S^i and a set of hard constraints $\phi_H^i \triangleq \phi_H \wedge \bigwedge_{j=1}^{i-1} \mathtt{CNF}(f_j = M_j)$.

Algorithm 1 shows a general schema of the BLO algorithm based on MaxSAT. Given a set of hard constraints ϕ_H and a list of optimization functions f_i, $i \in \{1, \ldots, k\}$, it first makes an initial check whether the problem has a solution (line 1). If yes, each function f_i is iteratively optimized using a MaxSAT oracle (line 5) and its optimal value is stored in M_i. The value M_i is then fixed (line 6) and the next function to optimize is considered.

For complex package management problems, e.g. with more than a couple of functions to optimize, current BLO-based solvers can require excessive run times. For instance, solving

Algorithm 1: General schema of BLO with MaxSAT

```
input  : $\phi_H$, $f_1, f_2, \ldots, f_k$
output : $(M_1, M_2, \ldots, M_k)$ if $\phi_H$ is satisfiable,
         false otherwise
1 if SAT($\phi_H$) = false then                  // initial check
2     return false
3 foreach $i \in \{1, \ldots, k\}$ do
4     $\phi_S^i \leftarrow$ CNF($f_i$)
5     $M_i \leftarrow$ MaxSAT($\phi_H \wedge \phi_S^i$)          // MaxSAT call
6     $\phi_H \leftarrow \phi_H \wedge$ CNF($f_i = M_i$)     // fixing value $M_i$
7 return $(M_1, M_2, \ldots, M_k)$
```

a problem can take more than 300 seconds[2], which is of course inadmissible in real world applications. The situation gets even worse for the case of weighted formulas.

A possible solution is to trade optimality for efficiency. One option is to use local search (e.g. [21]). Unfortunately, our attempts were unsuccessful, in part because of the many hard constraints that must be satisfied. As a result, we focus on approaches that guarantee that the hard constraints are still satisfied. For example, one can use specific MaxSAT algorithms that refine *upper bounds* on the optimal value. By doing so, after reaching a timeout one can stop the MaxSAT solver and get some upper bound, which can be treated as an approximation of the optimum value. However, the quality of the approximation done in such a way is typically low (see Section 4 for experimental results). Moreover, usually it is hard to even compute the first upper bound. Thus, it is not guaranteed that an instance will be solved within a given timeout. This leads us to a hybrid approach to the package upgradability problem, which is based on computing a subset-maximal set of satisfied soft constraints instead of computing a cardinality-maximum set of satisfied soft constraints. The method is described in Section 3.2.

3.2 Approximation. Hybrid Approach

As noted above, using MaxSAT solvers for finding exact (and even approximate) optima of individual optimization functions for complex package upgradibility problems is often not efficient. This section is devoted to a pragmatic approach to the considered problem, which performs very well on the most of the practical instances and provides high quality solutions. The idea of the method consists in combination of the BLO strategy described in Section 3.1 with efficient *approximate* algorithms for solving MaxSAT — namely, algorithms based on enumerating so-called *minimal correction subsets* (*MCSes*, e.g. see [22]).

Definition 1. *Given a set of hard clauses ϕ_H and a set of soft clauses ϕ_S, a set of clauses $\psi \subseteq \phi_S$ is called a minimal correction subset if*

1. *$\phi_H \cup \phi_S \setminus \psi$ is satisfiable;*
2. *$\phi_H \cup \phi_S \setminus (\psi \setminus \{c\})$ is unsatisfiable for any clause $c \in \psi$.*

An MCS enables us to approximate MaxSAT. An MCS set ψ gives us a subset of the soft clauses whose removal make the problem satisfiable. The smaller this set ψ is, the more soft clauses are satisfied and, thus, better approximation of MaxSAT is achieved. An important property of MCS is that it cannot be made smaller by removing some clauses. In contrast, MaxSAT unsatisfies the smallest possible number of soft clauses. Intuitively, MCSes correspond to *local optima* and MaxSAT provides us with the *global optimum*.

Hence, the cost of an MCS gives us an upper bound on the solution of the MaxSAT problem. For the case of unweighted formulas, the cost of an MCS is its size (the number of clauses in it); for weighted formulas — the sum of weights of all the clauses in the MCS. Recent work [23] proposes a number of efficient algorithms for computing one and enumerating all MCSes of unsatisfiable CNF formulas. These algorithms turned out to perform extremely well in practice.

There are different possible ways to apply computation of MCSes of a MaxSAT formula to the package upgradability problem. However, we mainly focus on the hybrid approach represented in Algorithm 2 while other possibilities are mentioned in Section 3.3.

Assume that `SAT`, `MaxSAT` and `BestApprox` calls can handle a given timeout and stop if it is exceeded. This can be done in a manner of line 4 of Algorithm 2. Note that Algorithm 2 is assumed to be able to asynchronously respect given timeouts and halt the execution of `MaxSAT` and `BestApprox` if necessary. Also assume that `CurrentTime`, `CNF`, and `ComputeApprox` calls can be done efficiently and do not take significant computational time. Similar to the BLO case, the arguments of Algorithm 2 are a set of hard constraints ϕ_H as well as a list of optimization functions $f_1, \ldots, f_k$. Note that one of the goals of the approach is to be able to stop and return a solution (or its approximation) within a reasonable timeout. Thus, the other two arguments are the timeouts Δ_{t_e} and Δ_{t_a} for the exact and approximate parts of the algorithm, respectively. In order to take into account the time used, Algorithm 2 first assigns current time to variable t_0 (line 1). Similar to the BLO case, the algorithm checks whether the hard constraints are consistent (see line 2). The lines 6–13 of the algorithm represent an implementation of BLO that takes into account a time limit. The algorithm tries to find exact optimal values for as many optimization functions as possible until it runs out of time (see line 10).

The next phase of the algorithm is approximation of the functions that were not optimized by BLO. The first function to approximate is the last one that was timed out in the BLO phase (see line 15 where j is assigned to be equal to i). During this phase of the algorithm, the second timeout value Δ_{t_a} gets used. We assume that calling a function `BestApprox` (see line 18) computes the cost M'_j of the *best MCS* found by enumerating all MCSes of the j-th formula withing the given timeout Δ_{t_a}. Observe that approximate value M'_j in some case can be equal to the exact optimum M_j. Variable $\mathcal{A}$ keeps the corresponding satisfying assignment of the formula. With each new function that is successfully approximated, value of $\mathcal{A}$ is updated. When the second timeout Δ_{t_a} is reached, assignment $\mathcal{A}$ is then used to compute approximate values for the functions that were not optimized nor approximated before (see line 25). Recall that calling function `ComputeApprox` is assumed not to take significant time. As a result Algorithm 2 produces a vector of found values $(M_1, \ldots, M_{i-1}, M'_i, \ldots, M'_k)$, where each M_i is an exact optimal value while M'_j can be both exact and approximate value of the corresponding function.

[2]See values of the timeout used in the latest MISC competitions [12].

```
Algorithm 2: Hybrid approach to BLO
input : $\phi_H$, $f_1, f_2, \ldots, f_k$, $\Delta_{t_e}$, and $\Delta_{t_a}$
1  $t_0 \leftarrow$ CurrentTime
2  if SAT($\phi_H, t_0, \Delta_{t_e}$) = false then        // initial check
3      return false
4  else if CurrentTime $- t_0 > \Delta_{t_e}$ then
5      return false
6  $i \leftarrow 1$
7  while $i \leq k$ do
8      $\phi_S^i \leftarrow$ CNF($f_i$)
9      $M_i \leftarrow$ MaxSAT($\phi_H \wedge \phi_S^i, t_0, \Delta_{t_e}$)        // MaxSAT call
10     if CurrentTime $- t_0 > \Delta_{t_e}$ then
11         break
12     $\phi_H \leftarrow \phi_H \wedge$ CNF($f_i = M_i$)        // fixing value $M_i$
13     $i \leftarrow i + 1$
14 $t_0 \leftarrow$ CurrentTime
15 $j \leftarrow i$
16 while $j \leq k$ do
17     $\phi_S^j \leftarrow$ CNF($f_j$)
18     $(\mathcal{A}, M'_j) \leftarrow$ BestApprox($\phi_H \wedge \phi_S^j, t_0, \Delta_{t_a}$)
19     if CurrentTime $- t_0 > \Delta_{t_a}$ then
20         break
21     $\phi_H \leftarrow \phi_H \wedge$ CNF($f_j = M'_j$)        // fixing value $M'_j$
22     $j \leftarrow j + 1$
23 while $j \leq k$ do
24     $\phi_S^j \leftarrow$ CNF($f_j$)
25     $M'_j \leftarrow$ ComputeApprox($\phi_S^j, \mathcal{A}$)
26     $j \leftarrow j + 1$
27 return $(M_1, \ldots, M_{i-1}, M'_i, \ldots, M'_k)$
```

3.3 Other Approximation Strategies

In this section we list and briefly describe other possible ways of doing approximation in the package upgradability problem. The simplest option would be to enumerate MCSes (or compute one MCS) for a CNF-encoding of the first optimization function within a given timeout Δ_{t_a}. After that, approximate values of each function f_i, $i > 1$, can be computed by a `ComputeApprox` call in a way similar to line 25 of Algorithm 2. Another option is to use the approach described in Section 3.2 but without the exact part, i.e. it would approximate individually as many functions as possible until the timeout is reached. Another alternative is to divide the value of timeout Δ_{t_a} into several timeouts, each can be used for approximating one function. Here values of different timeouts are not necessarily the same — instead, one can use more time for approximating the most relevant criteria.

And finally, one could construct a weighted MaxSAT formula comprising all the optimization functions. As noted above, in practice using a MaxSAT approach for such a complex optimization problem is proved to be not efficient. However, instead of calling a MaxSAT solver, one can, again, run an MCS enumerator. Each MCS found for the complex formula would approximate all the functions of the original problem.

Although there are many alternative ways of obtaining approximate solutions of the package upgradability problem that are based on computing MCSes, this article emphasizes the hybrid approach described in Section 3.2. The strength of the hybrid approach is that if it *is* possible to find quickly the exact solution, it is returned. However, for computationally hard problem instances, the approximate approach enables us to find a good approximate solution. Overall, the hybrid approach gives us the best of the two worlds, the exact and the approximate. Nevertheless, alternative approaches are also interesting from the user perspective and we return to this subject in Section 6.

4. EXPERIMENTAL EVALUATION

Experiments described in this section were performed on a large set of benchmarks from the *MANCOOSI International Solver Competition* 2012[3] (MISC-2012). The experimental results were obtained on an Intel Xeon 5160 3GHz, with 4GB of memory, and running Fedora Linux operating system. The experiments were made with a 800 seconds time limit and a 2GB memory limit.

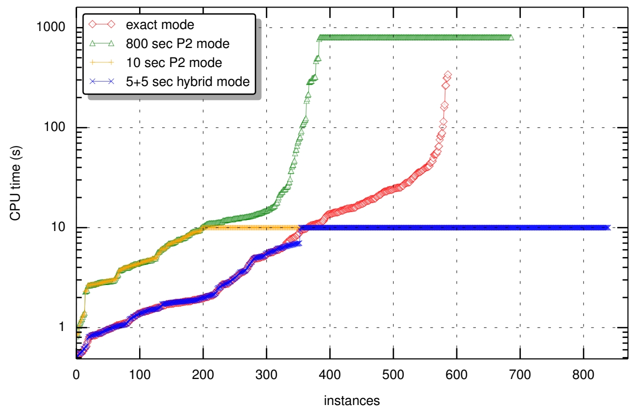
4.1 Experimental Methodology

The hybrid algorithm (Algorithm 2) was implemented on top of the tool *PackUp* [15]. *PackUp* provides a framework for solving the upgradability problem and its source code is available online under the GNU license [24]. The developed prototype is referred to as *PackUpHyb*. It accepts problem instances in the *Common Upgradability Description Format* (CUDF), which is a standard format of package description proposed within the framework of the MANCOOSI project.[4] The underlying SAT solver of the PackUpHyb solver is MiniSat 2.2 [25].

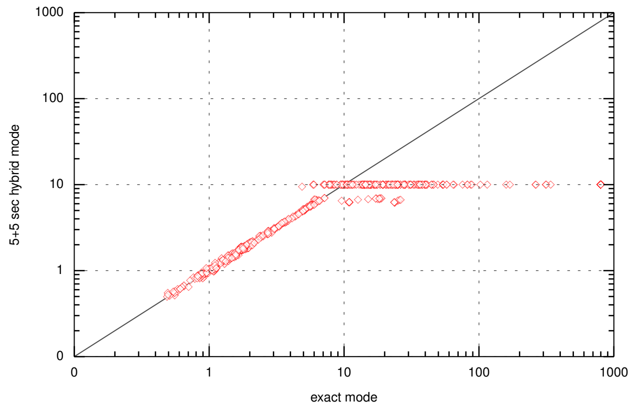
PackUpHyb has three general modes of operation. The first mode, called *exact mode*, makes use of a MaxSAT algorithm and does lexicographic optimization by finding the exact optimum for each optimization criterion of the package upgradability problem, as described in Section 3.1. The second mode, implements the hybrid approach described in Section 3.2, namely, Algorithm 2. Recall that the hybrid mode is parameterized by two different timeouts. The first timeout limits the time of the exact solving while the second one limits the time of the approximate solving. If the exact solver is not able to solve a problem within its timeout, it stops and the approximation process starts. The approximation is made for the first criterion that was not optimized by the exact solver.

The third mode is aimed at emulating the tool P2 [4], which is a well-known package dependency solver currently used in Eclipse IDE.[5] Similarly to P2, this mode of operation (hereinafter, we call it *P2 mode*) hinges on the idea of lexicographic optimization and, hence, optimizes all the criteria iteratively, as described in Section 3.1. Moreover, as an underlying solver intended for optimizing a criterion we use a pseudo-Boolean solver *EclipseP2*, which is a part of

[3] `http://www.mancoosi.org/misc-2012/`
[4] CUDF was developed in the MANCOOSI as a package description language with rigorous semantics and thus more amenable to scientific analysis. However, the language is as powerful as the package-description languages used in the linux distributions.
[5] `http://wiki.eclipse.org/P2`



(a) Performance of different modes of PackUpHyb



(b) Exact approach vs. Hybrid approach

Figure 2: Performance comparison

the Sat4j library and the basis of P2[6]. In case the EclipseP2 solver's execution process has timed out, it can produce an approximate solution of the problem. This enabled us to organize the comparison of the approximation quality between solutions found by the hybrid mode and the P2 mode.

The goal of the performed experiments is to show the advantages of the pragmatic approach proposed in Section 3 and implemented as the hybrid mode of PackUpHyb over the existing exact solver that participated in a series of the MISC competitions. First, we demonstrate the number of instances that can be solved by the considered approaches. Next, we compare the quality of the approximation produced by the P2 mode and the approach proposed in Section 3.

4.2 Experimental Material

The MISC-2012 competition has several benchmark categories, each is intended to emulate a real situation that can occur if a user makes a request to a package management system. Our experiments include all these categories. Each benchmark category represents a problem of optimizing a number of optimization criteria. The performance of the exact approach differs for different benchmark categories and depends on the optimization criteria used. There are 120 various CUDF files in the competition. Thus, for 7 categories the total number of package upgradability instances to solve is 840. Note that the size of the package universe (taking into accout unique package names, but not package versions) in the considered benchmarks varies from 27710 to 59094 with an average value 35275.5, which is close to the real repository size (around 40000 packages) in Debian-based Linux distributions. Therefore, the obtained performance results should mimic the behavior of the solvers on real world problem instances.

[6]Note that there is a specific variant of the P2 solver — P2CUDF — intended for solving package dependency problems given in the CUDF format. However, it is not presented in our experiments here. The reason is that for all considered instances it performs much worse than our mode of PackUpHyb that emulates P2. A possible explanation of this is that the SAT solver, which P2CUDF is based on, does not use the well-known *watched literals* and *VSIDS* heuristics (due to patent reasons).

Table 1: Number of solved instances

		Exact mode	hybrid mode	P2 mode
Timeout	5	282	282	126
	10	365	840	198
	800	584	840	684
	not solved	256	—	156

4.3 Comparing Performance

Figure 2a shows a cactus plot illustrating the performance of the considered modes of operation of the PackUpHyb prototype. Here it is convenient to use a logarithmic scale for the CPU time.

A typical user of a package management system is interested in having an answer to his request shortly. Therefore, it is essential for a package upgradability solver to be able to solve as many instances as possible in a short period of time. For this reason, we separately consider a 5 seconds timeout. In 5 seconds the exact and hybrid modes of PackUpHyb perform identically and are able to solve 282 instances (this is around 33.5% of the total 840 instances). Given the 5 seconds timeout, the P2 mode can cope only with 126 instances.

Considering a 10 seconds timeout, the exact mode can find a solution for 365 instances (including the 282 solved in 5 seconds) while the hybrid mode solves all 840 instances. Recall that the hybrid mode starts doing the approximation by enumerating local optima for the problem within 5 seconds and choosing the best of them. An interesting observation is that by doing this it is still able to find exact solutions (it does not reach the timeout) for 71 instances. The P2 mode can solve only 198 instances (around 23.5%) in 10 seconds.

Finally observe that for 256 instances (around 30.5%) the exact solver does not succeed in finding a solution within 800 seconds. Figure 2b shows a scatter plot that compares the runtime of the exact and hybrid modes of PackUpHyb. Given the 800 timeout, the P2 mode cannot find a solution (even approximate) of 156 instances (18.5%). Overall numbers of solved instances for different timeouts are represented in Table 1.

4.4 Approximation Quality

A solution of a problem instance is a tuple of integer numbers, each represents a value found for one of the optimiza-

Table 2: Quality of the approximation by the hybrid approach (%)

Category of benchmarks			Criterion index 1	2	3	4	5
	paranoid	avg.	0	2	–	–	–
		max.	0	82.7	–	–	–
	paranoid-size	avg.	0	2.1	-58.8	–	–
		max.	0	27.2	63.3	–	–
	embedded	avg.	2.4	0.9	–	–	–
		max.	27.2	15.3	–	–	–
	slowlink	avg.	10.5	482.5	-0.1	6.6	–
		max.	317	6614	9.3	178.6	–
	upgrade	avg.	0	0	0	0	–
		max.	0	0	0	0	–
	dist-upgrade	avg.	0	0	0	0	–
		max.	0	0	0	0	–
	trendy-size	avg.	0	2.6	38.7	0.3	-51
		max.	0	43.7	84.1	100	60

tion criteria. The quality of the approximation for a particular criterion is represented by a value of the approximation error, i.e. value $\frac{(v_a - v_e)}{v_e} \times 100\%$, where v_e is the exact optimum value and v_a is its approximated value. For instance, given tuples $(10, 20, 30, 40)$ and $(10, 25, 15, 40)$ that are an exact solution and an approximate solution, respectively, the approximation quality can be represented as a tuple $(0\%, 25\%, -50\%, 0\%)$. Note that due to the properties of lexicographic ordering there are situations when the approximate solution for a criterion j is better than the exact one (e.g. see $j = 3$ in the previous example), but only if there is a criterion i, $i < j$, such that its approximate value is worse than the exact value.

Also note that optimization criteria in the lexicographic optimization are sorted by their priority: the value of the first criteria is the most important to optimize while the last criterion is the least significant. Hence, it is preferable to have a smaller approximation error for the first criterion that is optimized.

Although the hybrid approach is able to solve all 840 instances, we can evaluate the approximation quality only for instances that are solved by the exact solver[7]. Also note that some of the instances might not have a solution. Thus, in this section we consider not 840 instances but only 525 that are solved by the exact approach and have a solution.

The number of optimization criteria vary from 2 to 5 for different categories of benchmarks. In our comparison of approximation errors for the hybrid and P2 modes we consider each optimization level separately. Since all benchmark categories have at least 2 optimization criteria, we consider 525 instances for the first two optimization levels. There are 386, 337 and 150 instances for the 3rd, 4th and 5th optimization levels, respectively. Cactus plots corresponding to approximation errors for optimization levels 1, 2, 3, 4, and 5 are shown in Figure 4a, Figure 4b, Figure 4c, Figure 4d, and Figure 4e, respectively. Again, the logarithmic scale is used in the plots.

[7]Since initially we did not aim at analyzing the approximation quality, the exact mode of the solver was run for 800 seconds only. However, getting more representative analysis of the approximation quality (with a greater value of the timeout and with the use of a computing cluster) will be an interesting topic of future research.

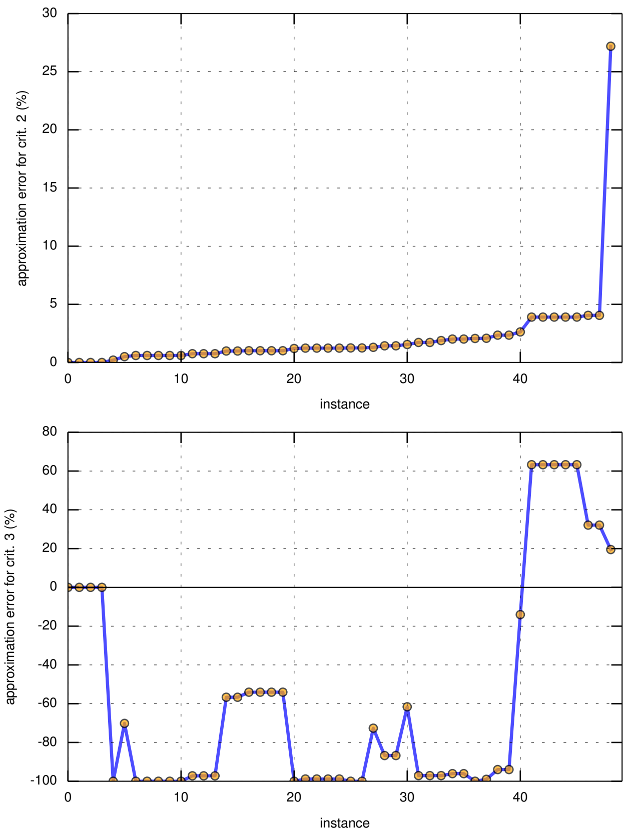
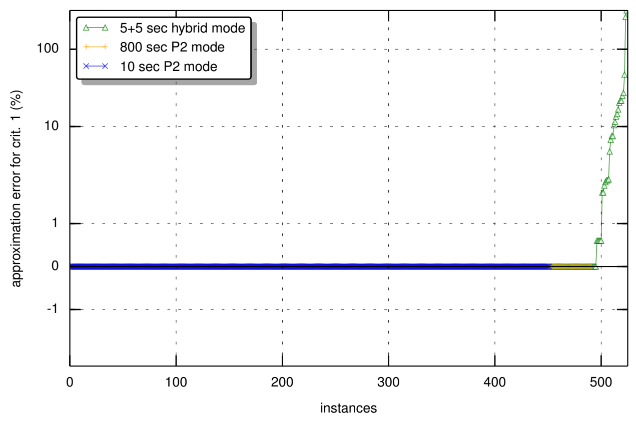


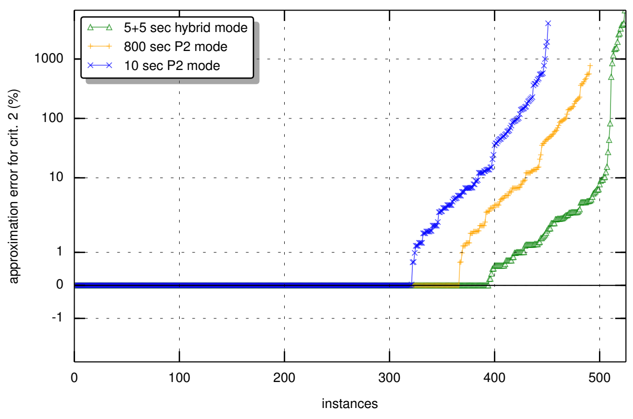
Figure 3: Quality of the approximation for the paranoid-size benchmarks category by the hybrid approach (instances are synchronized)

As one can see in the presented plots, not only the number of instances solved by the P2 mode is significantly smaller than the number of instances solved by the hybrid approach, but also the approximation error of the hybrid approach is orders of magnitude better than the approximation error of the P2 mode. Observe that this holds for both 10 and 800 seconds timeout for the P2 mode. Since the P2 mode solves fewer instances, we do not compare average approximation errors for the approaches. Instead, detailed information about the approximation error for the hybrid approach is shown in Table 2.

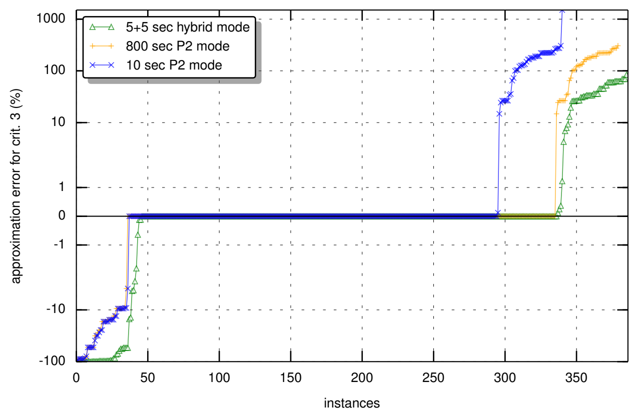
As an example, a detailed graph illustrating the approximation error for the *paranoid-size* category is shown in Figure 3. There are 49 instances in the paranoid-size benchmarks category and the number of optimization criteria is 3. Observe that the first optimization level is not presented in the figure since for all 49 instances an exact solution for the first level is found. Note that instances in the remaining 2nd and 3rd levels are synchronized in the figure so that one can see the quality of the full approximation for different instances. The maximal approximation error for the second criterion is 27.2% while the average is 2.1%. Values for the third optimization criterion are 63.3% and −58.8%, respectively. Observe that the negativity of the average value is caused by many instances for which the approximate solution is better than the exact one.



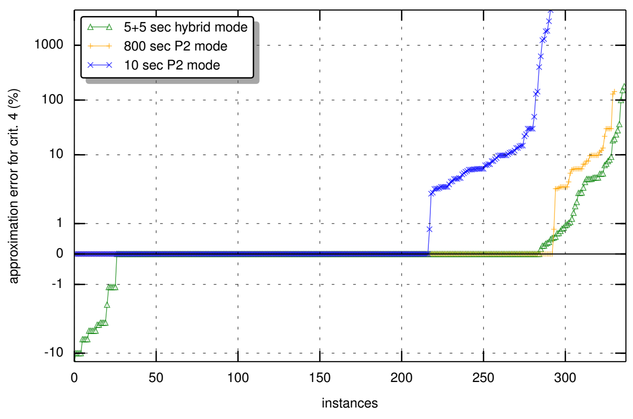
(a) Level 1



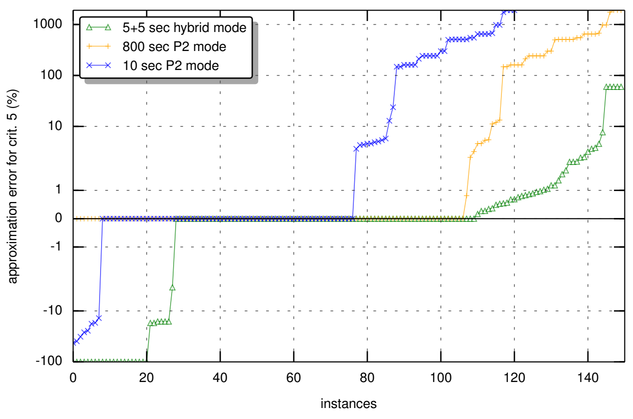
(b) Level 2



(c) Level 3



(d) Level 4



(e) Level 5

Figure 4: Comparison of approximation errors for the hybrid and P2 modes

5. RELATED WORK

The importance of SAT-based approaches to package management has been acknowledged in recent years [2–4, 15, 18, 20, 26]. Original work mapped the problem into a Constraint Programming formulation [3]. OPIUM [2] proposed optimization approaches for achieving better results, which are based on pseudo-Boolean optimization (PBO). Related work for the Eclipse IDE project includes [4]. Recent solutions, which allow for specifying a number of optimization criteria, are based on lexicographic optimization. A number of approaches have been proposed, which exploit a number of SAT-related technologies. These include inescp [20], Apt-PBO [18], PackUP [15], aspcud [5], and newer versions of P2 [4]. Of these, P2 [4] is the only tool that has been deployed and is being used by millions of users. Despite the importance of recent work towards added modelling flexibility and better quality solutions, computation of the optimum solution is unfeasible, because of the large required run times. As a result, deployed solutions return any solution identified within a short timeout, e.g. this is the case with P2. Furthermore, the importance of additional analyzes of software package repositories is illustrated by [26].

6. CONCLUSIONS AND FUTURE WORK

Managing package repositories poses a number of challenges. Indeed, it is not uncommon to encounter repositories with dozens of thousands of packages. Each of these packages represents a software artifact or component. And, each of these components may interact with other components. Consequently, when a user wishes to add a package to his or hers current configuration of packages, it must be done so with care. Namely, a package must not be installed if it conflicts with one of the other present packages and along with the package, all required packages must be installed. This, on its own, causes installation or uninstallation of packages to be a nontrivial computational task. This task becomes even more computationally challenging once we want to respect user preferences over the resulting package configuration. For instance, a typical user requires that the resulting package configuration differs from the original configuration as least as possible.

Altogether, package managers need to address two types of objectives: 1) Respect the requirements and conflicts of the packages. 2) Try to optimize for user criteria as best as they can. Recently, a number of approaches for exact optimization of user criteria have been developed. However, such approaches do not have guaranteed runtime. Indeed, it is not uncommon for a solver to need hundreds of seconds to find an optimal solution. Unfortunately, users are not willing to wait that long.

This paper proposes a technique of how to enable an exact solver to provide good approximate solutions whenever it is running out of time. The proposed technique hinges on the concept of *minimal correction sets (MCS)*. Intuitively, an MCS provides us with a *local* optimum, i.e. a better solution might exist but larger change to the package configuration would be required. Computationally, computing MCSes is far more advantageous than computing the global optimum. In practice this means that an MCS can be computed within seconds for instances where the global optimum is not found for hundreds of seconds.

Using MCSes contrasts with the approximation approaches used up to date. There, a complete solver would simply output the best solution that it has found so far. Such approach does not give any guarantee, while in MCSes we have the guarantee of local optimality. This advantage is indeed confirmed by our experimental evaluation, which shows that using the approximation based on MCSes enables us to solve a larger number of instances and with better proximity to the optimum than the "best so far" method.

This paper implements and evaluates a *hybrid approach*, which uses a MaxSAT solver for computing exact optima and MCSes for approximation. The MaxSAT solver is used for a given period of time to optimize the most important criteria and then approximation is used. This enables us to give a response within 10 seconds. Further, these responses do not deviate too much from the optimum.

It should be stressed that the proposed technique is *not* limited to the use of MaxSAT solvers. *Any* existing exact solver can be combined with our approximation technique in the same way. As such, the technique enables an exact solver to provide responses of good quality even in very short timeouts.

The use of an MCSes opens a number of avenues for future work. Several alternative way to integrate the approximation where already outlined Section 3.3, e.g. the total timeout can be distributed differently between the exact and approximate calculation. Beyond these alternatives, other applications can be envisioned. For instance, one could start enumerating MCSes and order them by a criterion different than a lexicographic one. This would allow considering for instance criteria like *leximin* [27]. Leximin is extremely interesting from the user perspective because it does not require the user to pick an order on the different criteria considered. And indeed, it is often hard for the user to prioritize one criterion over another. Instead, the user wants all these criteria to be all "good at the same time", which is made more precise by the leximin order. To our best knowledge, there is no publicly available package upgradability tool supporting leximin or any similar order. It is also expected that computing the optimum in the leximin order will be computationally much more difficult than for the lexicographic order. Yet, the MCS technique enables integrating any criterion at low computational cost.

There are also several topics of interest not covered by this paper. Although the P2 solver is heavily used in practice for the Eclipse IDE package management system [6], which maintains dozens of thousands of packages, there are other state-of-the-art approaches for package dependency solving (e.g. OPIUM [2], Aspcud [28]). Thus, an interesting subject of future work is the comparison of the developed method to such approaches as well as to the known software tools widely used by millions of Linux users every day, namely, APT for Debian-based distributions [29] and ZYpp for openSUSE [30]. ZYpp, which is based on a modern SAT solver, is one of the first success stories of applying SAT technology to package management in popular Linux distributions.

7. ACKNOWLEDGMENTS

This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), FCT grants ATTEST (CMU-PT/ELE/00-09/2009) and POLARIS (PTDC/EIA-CCO/123051/2010), and INESC-ID's multiannual PIDDAC funding PEst-OE-/EEI/LA0021/2013.

8. REFERENCES

[1] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon, "Maintaining large software distributions: New challenges from the FOSS era," in *Proceedings of the FRCSS '06 workshop*, ser. EASST Newsletter, vol. 12. Vienna, Autriche: EASST, 2006, pp. 7–20.

[2] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal package install/uninstall manager," in *29th International Conference on Software Engineering, ICSE*, 2007, pp. 178–188.

[3] 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 *ASE*. IEEE Computer Society, 2006, pp. 199–208.

[4] 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*, ser. IWOCE '09. New York, NY, USA: ACM, 2009, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1595800.1595805

[5] M. Gebser, R. Kaminski, and T. Schaub, "aspcud: A Linux package configuration tool based on answer set programming," in *LoCoCo*, ser. EPTCS, C. Drescher, I. Lynce, and R. Treinen, Eds., vol. 65, 2011, pp. 12–25.

[6] "Eclipse project," http://www.eclipse.org/.

[7] C. Hoover, "A methodology for determining response time baselines," in *Int. CMG Conference*. Computer Measurement Group, 2006, pp. 85–94.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie Mellon University, Nov., Tech. Rep. CMU/SEI-90-TR-021, 1990.

[9] E. C. Bailey, "Maximum RPM: Taking the Red Hat package manager to the limit," http://www.rpm.org/max-rpm.

[10] J. R. O. Aoki, "Debian new maintainers' guide," http://www.debian.org/doc/devel-manuals#maint-guide.

[11] R. Treinen and S. Zacchiroli, "Common upgradeability description format (CUDF) 2.0," MANCOOSI, Tech. Rep. 003, Nov. 2009, http://www.mancoosi.org/reports/tr3.pdf.

[12] "Mancoosi international solver competition," http://www.mancoosi.org/misc.

[13] "Mancoosi software," http://www.mancoosi.org/software.

[14] C. M. Li and F. Manyà, "MaxSAT, hard and soft constraints," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 613–631.

[15] M. Janota, I. Lynce, V. M. Manquinho, and J. Marques-Silva, "PackUp: Tools for package upgradability solving," *JSAT*, vol. 8, no. 1/2, pp. 89–94, 2012.

[16] M. Ehrgott, *Multicriteria Optimization (2. ed.)*. Springer, 2005.

[17] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce, "Boolean lexicographic optimization: algorithms & applications," *Ann. Math. Artif. Intell.*, vol. 62, no. 3-4, pp. 317–343, 2011.

[18] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: solving the software dependency problem using pseudo-boolean optimization," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 427–436.

[19] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-criteria optimization in answer set programming," in *ICLP (Technical Communications)*, ser. LIPIcs, J. P. Gallagher and M. Gelfond, Eds., vol. 11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 1–10.

[20] J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault, "Solving linux upgradeability problems using Boolean optimization," in *LoCoCo*, ser. EPTCS, I. Lynce and R. Treinen, Eds., vol. 29, 2010, pp. 11–22.

[21] K. Smyth, H. H. Hoos, and T. Stützle, "Iterated robust tabu search for MAX-SAT," in *Canadian Conference on AI*, ser. Lecture Notes in Computer Science, Y. Xiang and B. Chaib-draa, Eds., vol. 2671. Springer, 2003, pp. 129–144.

[22] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[23] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov, "On computing minimal correction subsets," in *IJCAI*, F. Rossi, Ed. IJCAI/AAAI, 2013.

[24] "Package upgradability problem solver," http://sat.inesc-id.pt/˜mikolas/sw/packup/.

[25] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[26] J. Vouillon and R. Di Cosmo, "Broken sets in software repository evolution," in *ICSE*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE / ACM, 2013, pp. 412–421.

[27] S. Bouveret and M. Lemaître, "Computing leximin-optimal solutions in constraint networks," *Artif. Intell.*, vol. 173, no. 2, pp. 343–364, 2009.

[28] M. Gebser, R. Kaminski, and T. Schaub, "aspcud: A Linux package configuration tool based on answer set programming," in *Proceedings of the Second International Workshop on Logics for Component Configuration (LoCoCo'11)*, ser. Electronic Proceedings in Theoretical Computer Science (EPTCS), C. Drescher, I. Lynce, and R. Treinen, Eds., vol. 65, 2011, pp. 12–25.

[29] "Apt – Debian Wiki," https://wiki.debian.org/Apt, accessed: 2014-02-27.

[30] "Portal:Libzypp – openSUSE," http://en.opensuse.org/Portal:Libzypp, accessed: 2014-02-27.