OPIUM: Optimal Package Install/Uninstall Manager

Chris Tucker
UC San Diego
cjtucker@cs.ucsd.edu

David Shuffelton
UC San Diego
dshuffel@cs.ucsd.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Abstract

Common Linux distributions often include package management tools such as `apt-get` in Debian or `yum` in RedHat. Using information about package dependencies and conflicts, such tools can determine how to install a new package (and its dependencies) on a system of already installed packages. Using off-the-shelf SAT solvers, pseudo-boolean solvers, and Integer Linear Programming solvers, we have developed a new package-management tool, called `Opium`, that improves on current tools in two ways: (1) `Opium` is complete, in that if there is a solution, `Opium` is guaranteed to find it, and (2) `Opium` can optimize a user-provided objective function, which could for example state that smaller packages should be preferred over larger ones. We performed a comparative study of our tool against Debian's `apt-get` on 600 traces of real-world package installations. We show that `Opium` runs fast enough to be usable, and that its completeness and optimality guarantees provides concrete benefits to end users.

1 Introduction

Dynamic software linking is pervasive, ranging from dynamic linking of libraries at runtime to inter-process invocation. Dynamic linking has numerous benefits, including saving memory both on disk and in RAM (since one copy of a library/package can be shared across many different applications), and allowing installed applications to easily benefit from updated libraries/packages. With these benefits, however, comes a configuration management problem that is difficult to solve. Libraries and software packages have dependencies that must be satisfied, and conflicts that must be avoided, otherwise the entire system, not just a single application, may become unstable.

In the context of Windows, this configuration management problem has lead to what is called "DLL hell": an application is installed with a variety of dynamically linked libraries, some of which override older versions of those libraries. Previously installed applications then break, because they were not meant to work with the new libraries. Users must typically intervene manually in order to bring the system back to a stable state.

In the context of Linux- and Unix-based systems, a variety of automated tools have been developed to address this configuration management problem, for example `apt-get` [14] on Debian, `yum` [4] on RedHat, and `fink` [1] on Mac OS. Using information about package dependencies and conflicts, such tools can determine how to install a new package, along with all its dependencies, on a system of already installed packages. However, because of the complexity in the dependencies and conflicts, such tools typically use heuristics and are therefore incomplete, in that even if a package is installable, the tool may fail to find a solution. Furthermore, if there are multiple ways of installing a given package, current tools arbitrarily pick between them without taking any user preferences into account. Such preferences could for example include picking smaller packages if the user has limited download bandwidth, or newer packages if the user wants the newest possible system.

Our goal in this work was to develop a uniform and complete solution to the configuration management problem that arises from having various inter-depending packages installed on the same system. In particular, using off-the-shelf SAT solvers, pseudo-boolean solvers, and Integer Linear Programming solvers we have designed a tool called `Opium` that solves the configuration management problem, and addresses the above limitations of existing package installers: it is complete (in that if there is a solution, it will find it) and it also allows one to optimize a given objective function. In addressing these limitations, `Opium` provides the following benefits:

- It improves the reliability of `apt-get`. Our measurements on 600 traces of real-world install attempts will show that about 23.3% of Debian users will be affected by `apt-get`'s incompleteness at some point in the lifetime of their system. This is especially concerning for companies like Linspire (where two of the authors worked) and distributions like Ubuntu, which are trying to make Linux usable by non-experts who don't have the sophistication to manually install packages if `apt-get` fails. The `Opium` tool entirely removes these incompleteness failures.

- `Opium` allows users to state their preferences through an objective function, and guarantees that this objective function will be minimized. This can in turn have real economic impact for Linux distributors. For example, the Linspire company provides a Linux distribution that is a low-cost alternative to commercial platforms like Microsoft's Windows and Apple's OS X. Their Linux distribution is therefore popular in many environments where bandwidth is at a premium (and often charged for per-byte). In order to provide the best experience at the lowest cost for the end user it becomes essential that bandwidth not be wasted. In this context, minimizing the size of packages delivered has the potential to offer a real economic benefit, while simultaneously reducing wait times for users. In our measurements, for example, we found a real-world install attempt where `apt-get`'s solution required downloading 129MB more than `Opium`'s optimal solution.
- There are cases where some packages need to be removed from the system before a new package is installed. Because `Opium` minimizes the number of packages being removed, it can find solutions that remove far fewer packages than existing package managers. In our experiments, we discovered a real user trace where an install attempt for OCaml using `apt-get` caused 61 packages to removed, including the Linux kernel. This poor user would not be able to reboot their machine after installing OCaml. Because `Opium` minimizes the number of packages being removed, it was able to find a solution that removed only 22 packages, none of which were the kernel.
- By providing a completeness guarantee, `Opium` allows Linux providers like Linspire to make quality of service claims regarding the predictability of user systems. In particular, if Linspire uses a tool like *debcheck* [10] to check the consistency of a given distribution (which essentially involves making sure that all possible packages in the distribution are installable), then they can provide the guarantee that all install attempts using `Opium` from that distribution will succeed on any user system.

Concretely, we have investigated three problems in the context of package management. In particular, given a set of installed packages, and information about package dependencies and conflicts, the three problems are:

> ***Install Problem*** : Determine if a new package can be installed, and if so, determine how.
>
> ***Minimum Install Problem*** : Determine the optimal way to install a new package, where optimality is determined by an objective function whose value is to be minimized.
>
> ***Uninstall Problem*** : Given a new package to install, determine the minimal number of packages (possibly none) that must be *removed* from the system in order to make the package installable.

The main contribution of this paper is solutions to the above three problems. We solve the *Install Problem* by running a SAT solver on a propositional encoding of the distribution (Section 3.1). This encoding is similar to, but independently developed from, the one presented in a forthcoming paper [10]. Further, we show how the SAT problem can be extended with an objective function, thus becoming a so-called *pseudo-boolean* problem that solves the *Minimum Install Problem* (Section 3.2). We also show how a well-known translation can be used to generate an Integer Linear Programming (ILP) problem from the pseudo-boolean problem. Highly tuned solvers exist for both pseudo-boolean problems and ILP problems.

We show how a SAT solver that produces a proof of unsatisfiability can be used to solve the *Uninstall Problem* (Section 3.3). Intuitively, if a package is not installable, from the proof of unsatisfiability of the SAT problem, we can determine what packages caused the conflicts, and therefore need to be removed.

We have implemented all of the above techniques in a tool called `Opium` (Optimal Package Install/Uninstall Manager) for installing packages on the Debian system. `Opium` uses Pueblo [13] for the pseudo-boolean solver, the GNU Linear Programming Kit (`GLPK`) [3] for the ILP solver, and the `foci` [11] theorem prover for producing unsatisfiability proofs. To evaluate the practicality and benefits of our algorithms, we performed a comparative study of `Opium` versus Debian's installer, `apt-get`, using 600 traces of real world installations (Section 4). Gathering information about the runtime and results of `apt-get` versus various configurations of `Opium`, we were able to quantify the benefits that `Opium`'s completeness and optimality provide, as well as show that it runs well within the limits of usability.

2 Overview

We begin with an overview of the install and uninstall problems and our solutions. A typical Linux distribution comprises a set of packages, each of which has a name and a version, distributed either on disk, or typically stored on online repositories. Each user has a subset of packages installed on their machine. Many packages depend on other packages to provide some functionality. For example the `apache` webserver may require the system to also have a `perl` interpreter. Thus, each distribution contains a meta-data file that explicates the requirements of each package of the distribution. For example, the meta-data for the `apache` package in the Debian distribution `sid` is shown in Figure 1:

```
Package: apache
Architecture: i386
Version: 1.3.34-2
Provides: httpd-cgi, httpd
Depends: libc6(>=2.3.5-1),
 libdb4.3(>=4.3.28-1),
 debconf(>=0.5) | debconf-2.0,
 apache-common(>=1.3.34-2),
 perl(>=5.8.4-2)
Conflicts: apache-modules,
 jserv(<=1.1-3)
 libapache-mod-perl
Description: HTTP server.
```

Figure 1: Rule for `apache`

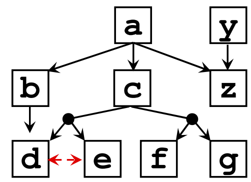


Figure 2: Distribution Graph.

Distribution Rules	Constraints
`Package: a`	
`Depends: b,`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{b}})$
`c,`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{c}})$
`z`	$(\neg x_{\mathtt{a}} \vee x_{\mathtt{z}})$
`Package: b`	
`Depends: d`	$(\neg x_{\mathtt{b}} \vee x_{\mathtt{d}})$
`Package: c`	
`Depends: d	e,`	$(\neg x_{\mathtt{c}} \vee x_{\mathtt{d}} \vee x_{\mathtt{e}})$
`f	g`	$(\neg x_{\mathtt{c}} \vee x_{\mathtt{f}} \vee x_{\mathtt{g}})$
`Package: d`	
`Conflicts: e`	$(\neg x_{\mathtt{d}} \vee \neg x_{\mathtt{e}})$

Figure 3: Fragment of Distribution Meta-Data and Corresponding Constraints

The meta-data contains details like the name, version, size, a description of the functionality provided by the package, *etc*. More importantly, it contains *depends* and *conflicts* clauses that stipulate which other packages should be on the system. The *depends* clauses stipulate which other packages *must* be present. Thus, in order to install `apache`, several other packages including `perl`, `libc6`, `libdb` and `apache-common` must be installed. Sometimes, a package requires any of a set of packages to be installed, possibly because each package in the set provides the required functionality. For example, the third depends clause is a disjunction that stipulates that *either* `debconf` (with a version greater than `0.5`) *or* `debconf-2.0` must be present. The *conflicts* clauses stipulate which other packages *must not* be present. Thus, the `apache` package should only be installed on a system that does not also have the `apache-modules` package, any instance of the `jserv` package with version less then `1.1.3` and so on. Thus, to install `apache`, the package manager must find out which other packages must be installed such that at the end, the system contains a set of packages that meet all the requirements specified in the distribution meta-data file.

We now illustrate our approach using a small distribution with the 9 packages `a`,`b`,`c`,`d`,`e`,`f`, `g`, `y`, and `z`. A distilled version of the meta-data rules for this distribution is shown on the left in Figure 3. In order for the package `a` to be installed on the system, packages `b`, `c` and `z` must also be installed, while for package `c` to be installed, one of `d`, `e` must be installed and one of `f`, `g` must be installed. The conflicts clause for `d` says that `e` must not be present on the same system as `d`.

Figure 2 shows a graph representation of the depends and conflicts clauses. Each package is shown in a square vertex, and there are directed edges to the other packages that must also be present. Whenever there is a disjunction in the depends, we represent it with a circle vertex which has directed edges to each package in the disjunction. Finally, there is a dotted edge between pairs of conflicting packages.

Installation Profiles. We call the set of packages installed on a machine the *installation profile* of that machine. A valid installation profile is one which meets all the depends and conflicts clauses of all the packages. Thus, the profiles $\{\}$, $\{\mathtt{y}, \mathtt{z}\}$ and $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{f}, \mathtt{z}\}$ are all valid installation profiles, as each package's depends and conflict clauses are met. On the other hand, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{z}\}$ is not a valid profile, as `c` requires one of `f` or `g` to be present, but both are absent from the profile. Similarly, the profile $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{z}\}$ is not a valid profile as it contains both `d` as well as a conflicting package `e`.

2.1 The Install Problem

Consider a user with the installation profile $\{\mathtt{z}\}$ who wishes to install the package `a`. The *install problem* is to determine whether there is some set of new packages *including* `a` that can be added to the machine, such that the resulting set of packages is a valid installation profile.

A tool like `apt-get` proceeds by traversing the dependency graph, and building up the set of other packages that must be installed before `a`. To be efficient it restricts the number of backtracks performed due to conflicts, and thus loses completeness, in the sense that `apt-get` may incorrectly report that there is no suitable set of new packages even though one exists.

Encoding Distributions as Constraints. Our approach to the problem is to encode it as a system of propositional constraints over variables representing the packages of the distribution. We create propositional variables for each package of the distribution and then create propositional constraints over the variables for each rule in the distribution. Every satisfying assignment to the constraints is such that the variables that get assigned TRUE form a valid installation profile for the distribution.

We create a variable x_p for each package p in the distribution. Next, we create constraints for each clause of the distribution. For instance, the first depends clause for a gets encoded as $(\neg x_{\mathtt{a}} \vee x_{\mathtt{b}})$ which stipulates that either $x_{\mathtt{a}}$ is false, *i.e.*, a is not in the profile or if it is, then $x_{\mathtt{b}}$ is true, *i.e.*, b is in the profile. The first disjunctive depends clause for c gets translated to: $(\neg x_{\mathtt{c}} \vee x_{\mathtt{d}} \vee x_{\mathtt{e}})$ which ensures that either $x_{\mathtt{c}}$ is false, *i.e.*, c is not in the profile, or if it is, then one of $x_{\mathtt{d}}$ or $x_{\mathtt{e}}$ must be true, *i.e.*, one of the packages d or e must also be in the profile. The conflicts clause for d gets translated to: $(\neg x_{\mathtt{d}} \vee \neg x_{\mathtt{e}})$ which ensures that both $x_{\mathtt{d}}$ and $x_{\mathtt{e}}$ are not true, *i.e.*, that both are not in the profile. In Figure 3, each row has a distribution rule in the left column and its propositional encoding in the right column.

SAT-based Installation Checking. To determine whether there is some set of new packages including a that the user can install that results in a valid installation profile, we use a SAT solver to find a satisfying assignment to the following *install formula*: $(\mathsf{Distrib}(R) \wedge x_{\mathtt{z}} \wedge x_{\mathtt{a}})$ which is the conjunction of $\mathsf{Distrib}(R)$, *i.e.*, the conjunction of all the constraints generated by the distribution (the right column in Figure 3), with the literals corresponding to the currently installed packages and the package to be installed.

For every satisfying assignment to the above formula, the set of packages corresponding to variables assigned TRUE is a set of packages including a that is a valid installation profile. It is easy to check that the assignment that sets all the variables other than $x_{\mathtt{g}}$ and $x_{\mathtt{y}}$ to TRUE satisfies the formula, and from it, we obtain a set of new packages including a that the user can download and safely install.

2.2 The Minimum Install Problem

In our example, there are actually two distinct satisfying assignments for the formula, and thus, two ways to safely install a. In the first one, described above, we add all the packages except g and y. Alternatively, we may install g instead of f as either one satisfies the depends clause for c. There are many situations where we would like to bias the package manager towards a particular choice – for example, towards the fewest number of new packages or the packages with the smallest total size. The *minimum install problem* is to find, given a *cost* for each package of the distribution, the set of new packages that must be installed with the smallest total cost.

The incompleteness of previous techniques makes it impossible to exhaustively search the solution space to find the set of packages with the minimum total cost. We extend our technique to the minimizing problem, by using *pseudo-boolean* (or equivalently, integer linear) constraints to encode the problem, and then using an appropriate solver to find the best solution.

Suppose that packages f and g have sizes of 5 and 2 MB respectively and all the other packages have size 1 MB. Consider a user with the profile $\{\mathtt{z}\}$ who wishes to download the fewest total number of bytes required to install the package a. To find the set of packages that the user should install, we generate and solve the pseudo-boolean constraint:

$$\begin{array}{ll} \textbf{min} & x_{\mathtt{a}} + x_{\mathtt{b}} + x_{\mathtt{c}} + x_{\mathtt{d}} + x_{\mathtt{e}} + 5x_{\mathtt{f}} + 2x_{\mathtt{g}} + x_{\mathtt{y}} + x_{\mathtt{z}} \\ \text{s. t.} & \mathsf{Distrib}(R) \wedge x_{\mathtt{z}} \wedge x_{\mathtt{a}} \end{array}$$

which specifies the satisfying assignment to the install formula, with the minimum total sizes (where we interpret TRUE as 1 and FALSE as 0). It is easy to check that the minimum assignment is the one that assigns TRUE to all variables except f and y, thereby resulting in the installation of all the other packages.

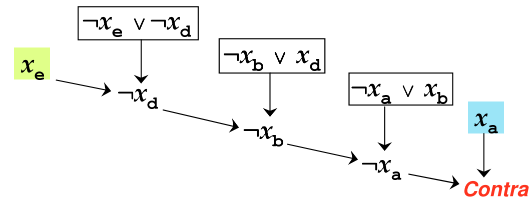


Figure 4: Resolution Proof of Contradiction of $\mathsf{Distrib}(R) \wedge (x_{\mathtt{z}} \wedge \mathtt{e}) \wedge x_{\mathtt{a}}$. Each leaf is a clause of the formula: the blue literal is from a, the package to be installed, the green literal is from the pre-existing (conflicting) package e, the white boxes are clauses from the distribution constraints. Each internal clause is generated by a resolution deduction of the form: $(A \vee x) \wedge (\neg x \vee B)$ implies $(A \vee B)$

2.3 The Uninstall Problem

Suppose that another user, with the installation profile $\{\mathtt{z}, \mathtt{e}\}$ wishes to install the package a. To do so, we must install b, and therefore d. Unfortunately, d is in conflict with a package e that is already installed. So, to install a we must first uninstall the previously installed package e that transitively conflicts with a. The *uninstall problem* is to find the set of packages currently installed on the system that must be removed in order to install some new package.

Using our technique, in order to determine if a could be installed, we would query a SAT solver with the install formula: $(\mathsf{Distrib}(R) \wedge x_{\mathtt{z}} \wedge x_{\mathtt{e}} \wedge x_{\mathtt{a}})$ The solver would report that the install formula was unsatisfiable, and would in addition return a *resolution proof tree*, such as that in Figure 4, which explained why the formula implied a contradiction and thus had no satisfying assignment.

The leaves of the proof tree correspond to clauses from the install formula. The leaf clauses that are the single variables obtained from previously installed packages yield the transitively conflicting packages that must be removed from the system to install the new package. Thus, in our example, the only leaf in the proof tree corresponding to a previously installed package is the $x_{\mathtt{e}}$ which reveals that e must be removed in order to install a. As with installation, there may be multiple sets of transitively conflicting packages, and so we show how to extend our technique to find the set that minimizes a given cost function.

3 Details

This section describes the details of our technique for solving package management problems using SAT solvers, pseudo-boolean solvers and ILP solvers. After first formalizing distributions and valid installation profiles, we formalize and present solutions to the three package management problems: the *Install Problem* (Section 3.1), the *Minimum Install Problem* (Section 3.2), and the *Uninstall Problem* (Section 3.3). Finally, we show how our solutions are combined in the tool `Opium` (Section 3.4).

Distributions

A *distribution* R is a finite set of *package rules*, where each package rule is a tuple of the form (p, D, C), where p is a *package* and:

- D is a set of *dependency clauses* for p that stipulate which packages must be present in order to install the package p. Each dependency clause is a disjunction of packages $p_1 \mid \ldots \mid p_k$. Intuitively, a dependency clause stipulates that *some* package from the set $p_1, \ldots, p_k$ must be present in order for the package p to work properly.
- C is a set of *conflict rules* for p that stipulate which packages must not be present on the same system as p. Each conflict clause is a package p' whose presence on the same system as p will cause problems.

For example, we formalize the distribution from Section 2 as the set of rules:

$(\mathtt{a}, \{\mathtt{b}, \mathtt{c}, \mathtt{z}\}, \emptyset), (\mathtt{b}, \{\mathtt{d}\}, \emptyset), (\mathtt{c}, \{\mathtt{d} \mid \mathtt{e}, \mathtt{f} \mid \mathtt{g}\}, \emptyset)$
$(\mathtt{d}, \emptyset, \{\mathtt{e}\}), (\mathtt{e}, \emptyset, \emptyset), (\mathtt{f}, \emptyset, \emptyset), (\mathtt{g}, \emptyset, \emptyset), (\mathtt{y}, \{\mathtt{z}\}, \emptyset), (\mathtt{z}, \emptyset, \emptyset)$.

Valid Installation Profiles

An *installation profile* for a distribution is a subset of the packages of the distribution, which could, for example, be the set of packages from the distribution installed on a particular machine. To ensure the proper functioning of the machine, we require the installation profile of the machine to be *valid*, meaning that it meets the requirements of each package in the profile.

To formalize this notion of validity, we start by defining when dependency clauses and conflict clauses are satisfied. An installation profile satisfies a dependency clause $p_1 \mid \ldots \mid p_k$ for p iff either p is not present in the profile, or *some* package in the set $\{p_1, \ldots, p_k\}$ is present in the profile. An installation profile satisfies a conflict clause p' for p iff either p is not present in the profile, or p' is not present in the profile. A *valid installation profile* for a distribution is one that satisfies the dependency and conflict clauses of each package rule of the distribution.

Readers familiar with Debian may realize that we have simplified the definition of a distribution in several ways. First, areal Debian distribution is in fact the union of two pieces – a repository residing on a central server, and the actual packages installed on the user's machine, each of which is a set of rules. To simplify the presentation, we assume here that the repository includes the rules from the user's machine. Second, associated with each package is a *version*, and depends and conflicts clauses can refer to specific versions of packages. We assume for simplicity that the clauses have been expanded to include all the versions of a particular package that are included in a distribution. Third, the rules also have a *provides* clause stipulating the set of virtual packages provided by a package. We make these simplifications for brevity – our implementation `Opium` handles all these features.

3.1 The Install Problem

We now turn our attention to the problem of determining whether (and how) a new package can be installed on a machine upon which some set of packages from a particular distribution is already installed. This problem is formalized as follows:

Problem 1 (*Install Problem*) *Given a distribution R, an installation profile P and a new package p, does there exist a set of packages P' containing p such that $P \cup P'$ is a valid installation profile for R.*

If such a P' exists, we say that p *can be installed on* P – by adding the packages in P', we get a valid installation profile containing the new package p. If instead no such P' exists, then it is impossible to safely install p on the machine already containing P.

Recall that our algorithm for solving the install problem is to reduce it to a system of propositional constraints whose satisfying assignments correspond directly to valid installation profiles. We introduce one boolean variable x_p for each package p to represent the presence of p. Truth assignments for the variables then correspond to installation profiles: x_p is assigned true iff p is in the corresponding installation profile. Once the problem has been converted to a system of propositional constraints, we use a SAT solver to determine whether the constraints are satisfiable – if so, we can directly extract the P' from the assignment returned by the solver, if not, we conclude that the installation is not possible.

The first step in our algorithm is to generate the propositional constraints for a distribution R. Our procedure for

$$
\begin{array}{rcl}
\mathsf{Distrib}(R) & \equiv & \bigwedge_{r \in R} \mathsf{Rule}(r) \\
\mathsf{Rule}(p, D, C) & \equiv & \bigwedge_{d \in D} \mathsf{Depend}(p, d) \wedge \\
 & & \bigwedge_{c \in C} \mathsf{Conflict}(p, c) \\
\mathsf{Depend}(p, p_1 \mid \ldots \mid p_k) & \equiv & \neg x_p \vee \bigvee_{i=1\ldots k} p_i \\
\mathsf{Conflict}(p, p') & \equiv & \neg x_p \vee \neg p'
\end{array}
$$

Figure 5: Propositional Distribution Constraints

Algorithm 1 $\mathsf{Install}(R, P, p)$

$f := \mathsf{Distrib}(R) \wedge \bigwedge_{p' \in P} x_{p'} \wedge x_p$
match $\mathsf{SatSolve}(f)$ **with**
| UNSAT $\longrightarrow$ **return** IMPOSSIBLE
| SAT $(A) \longrightarrow$ **return** $\{p' \mid A(x_{p'}) = \text{TRUE}\} \setminus P$

doing so is shown in Figure 5. Given a distribution R, $\mathsf{Distrib}(R)$ returns a boolean formula corresponding to valid installation profiles for the distribution R, where:

- $\mathsf{Rule}(p, D, C)$ returns a boolean formula corresponding to installation profiles that satisfy the package rule (p, D, C). The first and second conjuncts respectively ensure that each of the dependency and conflict rules are satisfied by the installation profile.
- $\mathsf{Depend}(p, p_1 \mid \ldots \mid p_k)$ returns a boolean formula that ensures that if the package p is in the profile, then *some* package from the set $p_1, \ldots, p_k$ is also in the profile.
- $\mathsf{Conflict}(p, p')$ returns a boolean formula that ensures that either p or p' is not in the profile.

Our algorithm Install for solving the *Install Problem* is shown in Figure 1. Making use of the above Distrib procedure, it creates a boolean formula capturing valid installation profiles including packages P and p, and then invokes a SAT solver to find a satisfying assignment. If a satisfing assignment A mapping boolean variables to truth values is found, we return the set of packages whose variables are assigned to TRUE (minus those packages in P), and otherwise, we conclude that it is not possible to safely install the package p.

3.2 The Minimum Install Problem

Owing to the disjunctions in the dependency rules, there are often many ways to install a new package. In these situations, we would like a way to select the "best" possible installation path. One may for example want to find the installation path in which the fewest number new packages are added. Or, if the user is connected via a low-bandwidth link, one may want to find the installation path with the least

Algorithm 2 $\mathsf{MinInstall}(R, P, p, \mathit{Cost})$

$c := \sum \mathit{Cost}(p') \cdot x_{p'}$
$f := \mathsf{Distrib}(R) \wedge \bigwedge_{p' \in P} x_{p'} \wedge x_p$
match $\mathsf{MinPBSolve}(c, f)$ **with**
| UNSAT $\longrightarrow$ **return** IMPOSSIBLE
| SAT $(A) \longrightarrow$ **return** $\{p' \mid A(x_{p'}) = \text{TRUE}\} \setminus P$

number of downloaded bytes. We generalize these problems as follows.

Problem 2 (*Minimum Install Problem*) *Given a distribution R, an installation profile P, a new package p, and a cost function Cost mapping packages to an integer cost, find a set of packages P' containing p with a minimum value of $\sum_{p' \in P'} \mathit{Cost}(p')$, such that $P \cup P'$ is a valid installation profile for R.*

The cost function above encodes the requirements for the "best" install. Once we find the P' with the minimum cost, the user can install the additional packages in P', and thereby obtain a valid installation profile containing the new package p.

Our technique of reducing the installation problem to propositional constraints extends to the *Minimum Install Problem*. In addition to the propositional constraints, we create pseudo-boolean constraints representing the linear cost function, and employ a pseudo-boolean solver to find a minimizing assignment.

A *pseudo-boolean constraint* is a pair $(\sum_{x \in X} c_x \cdot x, f)$ where X is a set of propositional variables, each c_x is an integer, and f is a propositional formula over X. The *cost* of a truth assignment A for the variables X is $\sum \{c_x \mid A(x) = \text{TRUE}\}$. A *minimum cost satisfying assignment* to a pseudo-boolean constraint is an assignment A that satisfies f, whose cost is less than or equal to the cost of every other satisfying assignment of f.

Our algorithm MinInstall for solving the *Minimum Install Problem* is shown in Figure 2. Using the cost measure, it creates a pseudo-boolean constraint capturing valid installation profiles including P and p, and then invokes a pseudo-boolean solver to find a minimum cost satisfying assignment. If one exists, it is returned by the solver, and from it we extract and return the minimum cost valid installation profile containing P and p. If no such assignment exists, we conclude that it is not possible to safely install p.

An alternative approach to solving the *Minimum Install Problem* is to reduce the pseudo-boolean constraints into an ILP problem using a standard translation [6]. One can then use an off-the-shelf ILP solver to find the minimum P'.

3.3 The Uninstall Problem

In many configurations, a new package cannot be installed because of conflicting dependencies with other packages al-

Algorithm 3 $\mathsf{UnInstall}(R, P, p, \mathit{Cost})$

$P_0 := P$
$f := \mathsf{Distrib}(R)$
$X' := \emptyset$
repeat
  $X := \{x_p\} \cup \{x_{p'} \mid p' \in P\}$
  $X' := \mathsf{ConflictSatSolve}(X, f)$
  $P := P \setminus \{x_{p'} \mid x_{p'} \in X'\}$
until $X' := \emptyset$
$P_c := P_0 \setminus P$
$\mathit{Cost}' := \lambda p.$ if $p \in P_c$ then $-\ \mathit{Cost}(p)$ else 0
$P' := \mathsf{MinInstall}(R, P, p, \mathit{Cost}')$
return $P_c \setminus P'$

ready installed on the system. In this case, we must first *uninstall* the packages prohibiting the installation, before attempting to install the new package. We would like to find the smallest set of packages that must be removed in order to make the new package installable.

Problem 3 (*Uninstall Problem*) *Given a distribution R, an installation profile P, a new package p, and a cost function Cost, find a set of packages P' with a minimum value of $\sum_{p \in P'} \mathit{Cost}(p)$, such that p can be installed on $P \setminus P'$.*

Once a minimum P' is found, we can remove the packages in P' and then obtain an installation profile on which p can be installed. We can then apply the algorithm MinInstall to determine the best way to install the new package p on the system.

There are several candidate cost functions for the uninstall problem. By assigning the all packages a constant non-zero cost, we can ensure that the *least* number of installed packages is removed. Another function could assign higher costs to more important or more popular packages, thereby ensuring that these packages do not get uninstalled.

To solve the *Uninstall Problem*, we will use an enhanced SAT solver that tells us which of the currently installed packages in P are prohibiting the installation of p. This enhanced SAT solver will compute an *overapproximation* of the packages that must be removed, and then we will use the previously described MinInstall procedure to prune the overapproximation to obtain a minimal uninstall set P'.

The enhanced SAT solver we make use of is implemented by a procedure called ConflictSatSolve. Given a set X of propositional variables and a propositional formula f, the procedure $\mathsf{ConflictSatSolve}(X, f)$ returns the empty set $\emptyset$ if the formula $\bigwedge_{x \in X} x \wedge f$ is satisfiable, and otherwise returns a *minimal* set $X' \subseteq X$ such that $\bigwedge_{x \in X'} x \wedge f$ is also unsatisfiable. The ConflictSatSolve procedure can be implemented using well-known algorithms. In particular, one can easily extend any DPLL-based SAT solver to produce resolution proofs of unsatisfiability [8, 17]. The set X' can then be computed from the resolution proof, by collecting the set of leaves in the proof tree that correspond to literals in X. In our setting, the literals correspond to packages – the set X will be the set of installed packages together with the new package p that is to be installed. In this context, the set X' returned by ConflictSatSolve will be *transitive conflict packages* prohibiting the installation of p.

Our algorithm UnInstall for solving the *Uninstall Problem* is shown in Figure 3. First, we save the currently installed packages in P_0. Second, we call the ConflictSatSolve procedure with the constraints generated by the current packages P and the distribution. If the constraints are not satisfiable, we remove the transitive conflict packages from the current set P, and repeat until the all constraints are satisfiable (there are no transitive conflict packages), *i.e.*, until p can be installed on the remaining packages. At this point, all potentially transitively conflicting packages have been removed from P, and the over-approximated set of conflict packages is $P_c = P_0 \setminus P$. Third, we call MinInstall starting with the installation profile G to determine what packages can be "added back" to P (and therefore were not absolutely necessary to remove). For this step, we use a modified cost function where the transitive conflict packages P_c have the negation of their original cost, and all other packages have cost 0. The negation causes MinInstall to in fact maximize the transitive conflict packages that are added back to P. Thus, the transitive conflict packages not added back by MinInstall are the minimum set of packages that must be removed.

The astute reader would have observed that another way to attack the uninstall problem is avoid the loop in Algorithm 3 by setting P_c to the set of *all* packages in P_0, and then running MinInstall. However, we choose to use ConflictSatSolve to find the transitively conflicting packages for two reasons. First, the set is typically quite small, and so the optimizing problem sent to MinInstall is relatively simple – the alternative would require the pseudo boolean solver to find a solution to a more complex problem, one that involved non-trivial costs for many more packages. Second, with our current formulation, it is easy to make the algorithm *interactive*, where at each iteration of the loop, the user can be asked which of the transitively conflicting packages in X' she would like to be removed. We can then only remove those packages from P in the next line. This approach, which we leave for future work, allows the user more control over which packages should be removed, and has the flexibility of not requiring that a suitable cost function be designed *a priori*.

3.4 Putting it all together: `Opium`

Figure 4 shows how the above algorithms are combined in our tool `Opium`, which takes as input a distribution R, an in-

Algorithm 4 $\texttt{Opium}(R, P, Cost_I, Cost_U, p)$

$R := \mathsf{Slice}(R, P \cup \{p\}$
$P' := \mathsf{MinInstall}(R, P, p, Cost_I)$
if $P' \neq$ IMPOSSIBLE **then**
  Install the packages P'
else
  Uninstall the packages $\mathsf{UnInstall}(R, P, p, Cost_U)$
  Install the packages $\mathsf{MinInstall}(R, P, p, Cost_I)$
end if

stallation profile P, an install cost function $Cost_I$, an uninstall cost function $Cost_U$, and a new package p that the user wishes to install, and updates the user's system so that it has a valid installation profile containing p.

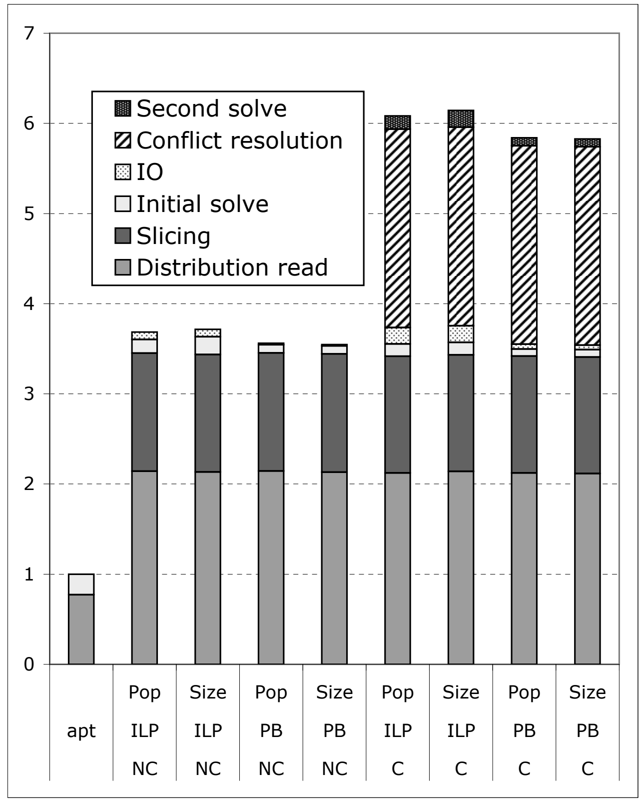
First, we *slice* the distribution rules with respect to the given installation profile and the package to be installed. Intuitively, the slicing procedure returns the subset of the input distribution rules that are relevant to the input packages. This procedure includes the rules of the input packages and transitively includes the rules of the packages the input package depends on or conflicts with. For example, slicing the distribution shown in Figure 2 with respect to the packages a, yields the package rules for all the packages except y. Without slicing, the times taken by `Opium` are about 15 times greater, taking several minutes to solve one problem, rather than several seconds.

Then, we call MinInstall to determine whether (without removing any existing packages), the new package can be installed. If there are no conflicts, *i.e.*MinInstall returns a set of new packages with the minimum install cost, and we download and install the new packages and return. If instead, MinInstall returns IMPOSSIBLE, then we call UnInstall to find the set of packages with the minimum uninstall cost, which are then removed from the system. Finally, we call MinInstall again, and this time it is guaranteed to find a set of new packages including p, which we download and install on the system. A simpler algorithm is to first call UnInstall as it would return the empty set if there were no conflicts. We choose to optimistically call MinInstall first as the majority of install attempts do not require uninstalls.

4 Evaluation

To evaluate the practicality of our algorithms, we performed a comparative study of `Opium` versus Debian's package installer, `apt-get`. The goal of this study was to quantify three measures: the running time of `Opium` versus `apt-get`, the amount of benefit provided by the completeness of `Opium`, and the amount of benefit provided by the minimization capabilities of `Opium`.

To perform our evaluation, we used 600 traces of real



world installation attempts collected by the servers at Linspire corporation. Each one of the 600 trace corresponds to a particular end user performing a series of installation attempts, and each installation attempt is a request to install a given package, which may in turn install/remove a variety of depending/conflicting packages. The 600 traces correspond to a total of 52,668 installation attempts, which amounts to an average of about 87 installation attempts per user.

We ran each installation attempt in 5 different ways. First, we used Debian's `apt-get`, which was the baseline for our comparison. Then we ran each installation attempt using `Opium` in four different configurations, varying the back-end (either a pseudo-boolean solver or an ILP solver), and the objective function (either minimize download size or maximize the popularity of installed packages). These experiments took about 24 hours to run using 100 nodes of the FWGrid cluster [2].

4.1 Runtime

Figure 4.1 shows the runtime of `Opium` normalized to the runtime of `apt-get`. To get a sense of the scale, the average runtime of `apt-get` was 3.14 seconds, and this shows up as a bar of height 1 in Figure 4.1. The rightmost eight bars of Figure 4.1 show the runtimes for `Opium`. The labels for these bars use the following abbreviations: (1) NC: no conflicts occurred *versus C*: conflicts occurred (2) ILP: ILP solver was used *versus* PB: pseudo-boolean solver was used (3) Pop: the objective function maximized popularity *versus* Size: the objective function minimized total download size.

Each bar shows inside of it the various contributors to the runtime: (1) *Distribution read*: time to read the distribution from disk into memory (2) *Slicing*: time to perform the slicing optimization described in Section 3.4 (3) *Initial solve*: time to perform the first call to MinInstall in the `Opium` algorithm from Section 3.4 (4) *IO*: time to write the pseudo-boolean or ILP problems to disk for the solvers to read, and time to read the results back from the solvers (5) *Conflict resolution*: time to perform conflict resolution, which is the call to UnInstall in the `Opium` algorithm (6) *Second solve*: time to run the second call to MinInstall in the `Opium` algorithm.

There are a variety of important points to get out of Figure 4.1:

- In the cases where there is no conflict resolution, which account for 84.3% of the install attempts, `Opium` is about 3.5 times slower than `apt-get`. In the remaining cases, `Opium` is about 6 times slower than `apt-get`. Although this may seem high, when taking into account the total time to run the installer *and* to download the required packages, on average, `Opium` is 34.0% slower than `apt-get` assuming a 300kBps cable modem connection, 11.2% slower on a 100kBps DSL line, and 0.2% *faster* on a 10kkBps dial-up modem (`Opium` is able to run faster on a modem because it optimizes for number of bytes downloaded, and so it downloads less bytes than `apt-get`).
- The dominant components of the `Opium` runtime are reading the distribution, performing the slicing optimization, and performing conflict resolution. The actual time to run Pueblo or GLPK accounts for only a very small proportion of the total runtime of `Opium`.
- The Pueblo solver runs about twice as fast as the GLPK solver, and it even runs slightly faster than the `apt-get` backtracking solving algorithm.
- The runtimes of install attempts that optimize size are very similar to the runtimes for attempts that optimize popularity, which is an indicator that the runtimes are unlikely to depend on the objective function.

There are further opportunities for optimizing the performance of `Opium` that we have not yet explored. One of them is the runtime it takes to read a distribution. Because our implementation of the `Opium` parser is naive, `Opium` takes about 3 times longer than `apt-get` to read and load a distribution in memory, something that can be fixed with further tuning. Another area where performance could be improved is conflict resolution. The ConflictSatSolve operation is currently implemented in a separate theorem prover, which incurs additional overhead. Furthermore, because ConflictSatSolve is called repeatedly on very similar problems, using an incremental SAT solver for implementing ConflictSatSolve would likely have a drastic impact on the performance of conflict resolution.

4.2 Completeness

To quantify the benefit provided by `Opium`'s completeness, we look at the number of times that `apt-get` fails to find a way of installing a package when in fact there is a solution (which `Opium` is guaranteed to find because it is complete). Out of the 52,668 install attempts, `apt-get` was not able to find a solution 357 times, and of these 357 cases, `Opium` was able to find a solution 322 times. The remaining 35 cases, on which both `apt-get` and `Opium` fail, are indications of bugs in the distribution (for example, one package in the distribution depending on another one that is not in the distribution).

These numbers show that `apt-get` fails to find a solution when one exists in about 0.61% of install attempts. This is not a large error rate, but one has to remember that users perform many install attempts over the lifetime of their system. Assuming an average of 87 install attempts over the lifetime of a user system (computed from the average size of our trace lengths), the chance that a user will hit an incompleteness error in the lifetime of their system can be computed to be 41.2%. The actual number collected in our experiments is smaller than this, but in the same ballpark: 23.3% of the 600 traces encountered an incompleteness limitation of `apt-get`. These numbers indicate that the completeness of `Opium` has the potential to improve the end-user experience of a large fraction of Debian users.

4.3 Minimization

We first evaluate the impact of `Opium`'s ability to minimize the number of packages that are removed from the system. On our traces, `Opium` removed less packages than `apt-get` in 209 cases out of the 52,311 install attempts where `apt-get` succeeded. This is a small percentage of all install attempts, but the impact in those cases can be significant. In 9 cases `apt-get` removed 10 packages or more than what was necessary, and the worst of these cases is the example mentioned in the introduction, where `apt-get` removed 61 packages, including the kernel, whereas `Opium` only removed 21 packages, none of which was the kernel.

We also evaluated the benefits of `Opium`'s ability to minimize the number of downloaded bytes. In about 4.4% out of the installation attempts where `apt-get` succeeded, Opium found a better solution than apt-get. Although this is only a small percentage of all install attempts, when there is a difference between the optimal solution and the apt-get solution, that difference on average is about 2MB, which is considerably large. There are also 7 install attempts in which Opium beat apt-get by over 100MB, and one case in which Opium beat apt-get by 129MB. In about 0.2% of the installation attempts, apt-get finds a better solution than Opium by an average of about 1.6MB. This happens despite Opium's optimality because apt-get sometimes removes more packages than Opium, and once these additional packages have been removed, it is possible that apt-get can find a better solution.

Another interesting measure to look at is how many downloaded bytes Opium saves over entire user traces. Summing the downloaded bytes over entire traces, we find that Opium beats apt-get on 95.9% of the traces by an average of 7.7MB (with a maximum of 185MB), and it matches or does better than apt-get on 98.4% of all traces. The most apt-get beats Opium by is 21MB, but it does so by removing 12 more packages than necessary.

5 Related Work

One line of work that is related to ours is the research done by the WP2 group inside the EDOS project. The broad goal of this group is to address issues relating to dependency management *on the repository side* [10], whereas our focus has been *on the client side*. In the context of helping repository builders, the WP2 group has implemented a tool called *debcheck* [10] that uses a SAT solver to check that a repository does not contain broken packages (i.e.: packages that cannot be installed). As the authors of *debcheck* write in [10], the problem of optimizing the installation of packages on a user machine, which Opium solves, "is a task radically different, and in principle much more difficult than verifying that a repository does not contain broken packages." In particular, our paper contributes beyond the work on *debcheck* in three ways, all of which are motivated by our focus on the client side of the problem: (1) our work adds the extra dimension of finding *optimal* solutions with respect to an objective function (2) in addition to solving the *Install Problem*, we also optimally solve the *Uninstall Problem* (3) we perform a comparative study of our tool against apt-get on real-world installation attempts.

Another project that is related to ours is the Smart Package Manager [12], which attempts to be complete and to find the best solution given a user policy. There is little documentation about the techniques used in Smart, and our investigation of the source code shows that it enumerates all possible solutions, which, as pointed out in [10], is prohibitively expensive.

More broadly, our work is also related to research projects that process dependencies automatically. In the context of static component-based software linking, tools exists for checking that dependencies between a given set of components are met, for example using typed interfaces [7, 5, 9]. Tools also exist for analyzing dependencies to optimize, debug, and test programs [15, 16] In contrast to these projects that check or analyze dependencies, our goal is to *discover* an optimal set of components that meet certain dependency requirements.

References

[1] fink. http://fink.sourceforge.net.

[2] FWGrid Project. http://fwgrid.ucsd.edu.

[3] GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk.

[4] Yum: Yellow dog Updater, Modified. http://linux.duke.edu/projects/yum.

[5] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE*, pages 187–197, 2002.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 1990.

[7] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[8] A. V. Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *7th International Symposium on Artificial Intelligence and Mathematics(AMAI)*, 2002.

[9] D. B. MacQueen. Modules for standard ml. In *LISP and Functional Programming*, pages 198–207, 1984.

[10] 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 *Proceedings of the International Conference on Automated Software Engineering (ASE 06)*, 2006.

[11] K. L. McMillan. An interpolating theorem prover. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, pages 16–30, 2004.

[12] G. Niemeyer. Smart package manager. http://labix.org/smart, 2006.

[13] H. M. Sheini and K. A. Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–96, 2006.

[14] G. N. Silva. APT Howto. http://www.debian.org/doc/manuals/apt-howto, 2005.

[15] 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)*, 1998.

[16] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *Proceedings of the International Conference of Automated Software Engineering (ASE 02)*, 2002.

[17] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE: Design Automation and Test Europe*, pages 10880–10885, 2003.