Dependency Resolution in PISI

Eray Özkural

July 21, 2005

1 Introduction

Dependency resolution in package management systems have a significance in that they are the key to providing system stability and internet upgrades. The scale of package databases requires the dependency resolution mechanism to be efficient and correct, motivating a closer look at the theory.

2 Review

Dependency resolution has been taken in the most general setting as the famous SAT problem of propositional logic. If we consider a system D of dependency statements D_i, each statement can be taken as a proposition in propositional logic which states, for instance:

D_i: if package a is installed or package b is installed, then package i is installable.

...

The system is thus understood as the conjunction of such facts, giving us a logical programming formulation to determine installation conditions.

Note that for simplicity we do not consider the nuances in upgrade and remove operations at the moment.

However, using a SAT solver for this operation may be shooting a fly with a bazooka. We observe that only certain forms of propositions will be necessary for a dependency system. Furthermore, as we shall see further constraints and optimizations may be required of the system that are not modelled well with the SAT problem.

We use a graph theoretic approach instead. A directed graph (digraph) $G = (V, E)$ is formally a set of vertices V and a set of edges E where each edge (u, v) represents an edge from a vertex to another. Topological sort of a graph gives a total ordering of the vertices in which there are only forward edges.

3 Package operation planning

The dependency resolution problem may be viewed as a simple forward chaining problem, where we would like to begin from an initial state S_0 and by following allowable system transitions $t_i : S- > S$, arrive at a desired system state S_f.

A system state S_i is defined as the set of installed packages on the system together with their versions, i.e. $S_i = \{(x, v) : x \text{ is installed}, v = version(x)\}$. An atomic system transition t_i chains one system state into another, making one ACID change on the system. The usual atomic transitions are the single package install and remove operations found in low-level package management code of PISI. Note that in PISI, an upgrade operation is identical to a remove operation followed by an install operation (which sets it apart from some other packaging systems).

A package operation plan is thus naturally conceived of as a sequence of atomic system transitions. Given an initial state and a final state, the job of the package operation planner is to determine whether there is a plan, and if so find the "best" one.

Where there are no versions involved (e.g. upgrade/downgrade), we will replace the pair (x, v) with x in the definitions for simplicity.

3.1 System consistency

It is worth mentioning here the concept of system consistency. As in a database transaction, it is not acceptable that the system violates an invariant afterwards. In the context of PISI, system consistency is composed of two conditions for the current set of installed packages.

1. All package dependencies are satisfied (we may call this a closed system)

2. No packag conflicts are present.

Therefore, by atomic transition we also mean one that does not corrupt system consistency. The system is thus never in an inconsistent state. We will explain the conflicts later, for the present let us look at the dependency condition.

3.2 Solving the simplest case with topological sorting

We will now concentrate on a simple form of the problem which can be solved with topological sorting. This form is not concerned with versions. From initial set of packages S_0, we would like to install in addition a new set A of packages obtaining $S_f = S_0 \cup A$.

The only relations considered are of the form: a Depends on b, or more briefly aDb.

The graph of all such simple dependency relations is a directed graph (digraph) G. For each dependency relation aDB, there is an edge $a \to b$ in G. Accessing graph G usually requires a database operation and is therefore expensive.

We now consider the digraph G_A of the minimal set of simple dependency relations which contains all information required to construct a plan to install packages A. G_A is a vertex induced graph such that the fringe of A, e.g. vertices with out-degree 0 are already installed. Vertices of G_A are taken from S_f. First, let us explain the labelling scheme. Already installed vertices are labelled with 'i'. Packages to be added are labelled with 'a', and packages to be installed due to dependencies are labelled with 'd'. We construct the graph as follows

```
 1: G_A ← isolated vertex set A labelled with 'a'
 2: repeat
 3:    done ← true
 4:    for each u ∈ V_A with out-degree 0 do
 5:       for  v ∈ adj(u) of G do
 6:          if vis ∉ V_A then
 7:             done ← false
 8:             if v is installed then
 9:                label v with 'i'
10:             else
11:                label v with 'd'
12:             end if
13:             add (u, v) to G_A
14:          end if
15:       end for
16:    end for
17: until done
```

By this iterative expansion, we do a minimum number of database accesses to G and construct a dependency graph in memory. If the G_A's fringe has vertices with non 'i'-labels, then A cannot be installed. Otherwise, we find a topological sort L of G_A, and in the reverse order, install packages for vertices labelled with 'a' or 'd'. Observe that, by definition of a topological sort, installing packages in the reverse order of a topological sort guarantees that no package is installed before all of its dependencies are installed. Thus, this yields a consistency-preserving plan.

3.3 Conflicts and COMAR dependencies

The tags `Conflicts` and `Provides` are inherited from Debian distribution. A conflict between two packages (a conflicts with b) is a symmetric relation

that prevents the packages (a, b) from being installed simultaneously. (It is sufficient that only one direction of the relation is declared, the other direction is inferred) Provision in the form of a provides A denotes that a implements a virtual package abstraction A.

In PISI, a package can provide an object of a COMAR Object Model (OM), and is currently the only model of a "virtual package". In the following example, let $a_1, a_2, \ldots, a_n$ provide the OM A. A package can depend on another package's OM, for instance b comar-depends on A (or in short form bDA). In this case, it is sufficient that only one of the a_i are installed. To resolve this, the user is asked to choose from a list of alternatives immediately, since otherwise there is unavoidable combinatorial explosion (in the form of having to consider $\Pi_{bDA} num(A)$ graphs in the worst case where $num(A)$ is the number of alternatives for comar OM A; the problem is that there seems to be no simple solution to solve satisfiability with arbitrary disjunctions in package dependency, short of a SAT solver).

4 Examples

4.1 A single package upgrade

plan: upgrade $(a, 1)$ to $(a, 2)$

rules:
$(a, 1)$ depends on $(b, 1), (c, 1)$
$(a, 1)$ conflicts with $(d, 1)$
$(a, 2)$ depends on $(c, 3), (d, 2)$
$(a, 2)$ conflicts with $(b, 1)$

initial state:
$(a, 1), (b, 1), (c, 1)$ installed

plan:
remove $(b, 1)$
remove $(c, 1)$
remove $(a, 1)$
install $(c, 3)$
install $(d, 2)$
install $(a, 2)$

4.2 A multi package remove

plan: remove $(a, 2), (b, 3), (c, 2)$

rules:
$(c, 2)$ depends $(a, 2)$
$(d, 2)$ depends on $(b, 3), (c, 2)$
$(e, 1)$ depends on $(a, 2)$
$(f, 2)$ depends on $(e, 1)$
$(g, 2)$ depends on $(e, 1)$

plan:
remove $(f, 2)$
remove $(g, 2)$
remove $(e, 1)$
remove $(d, 2)$
remove $(b, 3)$
remove $(c, 2)$
remove $(a, 2)$