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 (where S is the set of all states).

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 package 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 v ∉ 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 Dependency conditions

In the PISI specification, we allow a dependency to specify a local condition, for instance a program may require a dependency on `libx` with pardus

source release 3 or greater. Another program may require a dependency on a particular source release. These conditions are local because they can be computed over the elements of system state S_i, e.g. package (name, version) pairs. Let us denote this condition by a predicate $P(b)$ such that $aDbiffP(b)$. The predicate P for the dependency aDb can be stored as edge data for (u, v) on the graph.

In this case, the vertices of the package dependency graph G and the planning graph G_A retain the version information along with the package name. The dependency relation thus holds between two pairs (p_1, v_1) and (p_2, v_2), satisfying a given predicate $P(p_2, v_2)$. When constructing the graph, we therefore take this predicate into account and admit a new edge (u, v), and thus a new vertex v into G_A if and only if the target vertex satisfies $P(v)$.

3.4 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).

The resolution of conflicts to maintain system consistency condition 2 is easier to achieve. This can be always satisfied by disallowing installation of a package that would violate the condition. In the install operation, after constructing the partial dependency graph G_A, we merely have to check whether any conflict appears within the vertices of G_A. If so, then the operation is untenable, since G_A shows the future state of the installed system. Since a conflict is symmetric, it is represented as a bidirectional edge $a \leftrightarrow b$. To distinguish dependencies from conflicts, the edges would have to be labelled in this case, for instance with 'd' and 'c'.

3.5 Remove operation

Dependency resolution for remove operation is similar to install. The only difference is that we remove the packages in the topological order rather than installing packages in the reverse topological order.

4 Remote repositories and upgrade operation

The upgrade operation is more complicated. First of all, the system has to distinguish between the current relation graph (e.g. dependencies and conflicts), and the future relation graph which may be different in rather important aspects. In theory, we allow any dependency and conflict to change. Therefore, we have a G_0 which represents the current relations (among installed packages) in the system, and a G_f which is probably taken from a remote package repository. We begin by noting that G_0 and G_f have to be compatible. That is, to say, if a package (u, v) is shared across two times, then the declarations made by the package are one and the same.

G_0 can be calculated from the package information (e.g. metadata) of the installed packages and is stored by PISI in a dedicated database. G_f is most likely constructed from a PISI Index file corresponding to a particular package repository. Accessing both of these entities is expensive and we should take care to minimize access as in the previous section.

To preserve consistency during individual transitions, the planner can choose to remove a minimal number of packages from the system to bring it to a clean state, and then install the new versions of these packages in the correct order. Let us assume that it is indeed possible to achieve this "clean state". Apparently, this is not always possible because other packages may depend on the package(s) to upgrade. At any rate, to achieve this, first we need to calculate subgraphs of G_0 and G_f. We can calculate alternative plans from these subgraphs if need be.

Let A be the set of packages to be upgraded from a given repository. $G_{A,0}$ is the subgraph of G_0 induced by the "upgrade closure" of A. The "upgrade closure" of a set A of packages is defined as a minimal set of packages $B \supseteq A$ such that there is no package in B that requires an upgrade for A to be upgraded. This is found by assuming that the current system state S_0 is consistent, and by constructing a relation graph of the future state of the system to detect the dependencies that have changed.

Obviously, to make a plan, we must first know the goal state. In a multi-package upgrade, the exact details of the goal state depend on the graph G_f of the repository. Thus, we construct a graph $G_{A,f}$ that is a vertex-induced subgraph of G_f such that it contains all information relevant to upgrading packages A. We begin by a vertex induced graph by A. These are the packages that will be upgraded in any case. Then, we make a pass on the vertices, and look at all the outgoing edges, we compare whether

this edge has changed in any substantial way from the previous version. In particular, we are interested in whether the predicate of the edge is valid for the version of the same package in our current system. Every compared vertex in this manner is marked done, and the edges not valid for the current system pull new unmarked vertices into G_f, this continues until there are no unmarked vertices left. Hence, the vertices of G_f are the packages that must be upgraded.

To actually carry out the upgrade a strategy is to upgrade one by one all the packages in some order. A good order is again the reverse topological order order, in fact, the operation is merely a special case of a multi-package installation code that can install from a remote repository. However, in case no package depends on the packages to be upgraded, then we can carry out a completely consistency-preserving plan as discussed above.

5 Examples

5.1 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

In this case, we can find a consistency-preserving plan in terms of install and remove operations.

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

5.2 Another upgrade

objective: upgrade $(b, 1) \rightarrow (b, 2)$

current dep: $(a,1) \rightarrow [=1](b,1) \rightarrow [=1](c,1) \rightarrow [=1](d,1)$
repo dep: $(a,1) \rightarrow [=2](b,2) \rightarrow [=2](c,2) \rightarrow [=1](d,1)$

In this case, we cannot remove $(b,1)$ because it's locked in the chain. In fact, here there is no consistency-preserving plan in terms of atomic single package transitions: install, remove, upgrade. In these cases, it seems best to resort to upgrade in place, and in the reverse topological order of dependencies.

plan:
upgrade $(c,1) \rightarrow (c,2)$ upgrade $(b,1) \rightarrow (b,2)$

5.3 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 $(a,2)$