Dependency Hell

1 Background and Motivation

Package managers are key programs for not just operating systems, but any system that relies on packages maintained by different developers. The core function we will examine is updating a single package.

When you upgrade a package, the manager generates an upgrade plan before execution. The upgrade does two things. First, it determines if upgrading a given package is possible. Second, it determines the order of deletions and installations to safely upgrade the package.

Traditionally, package managers have used heuristics to create the upgrade plan, but recently there has been a shift towards using SAT solvers. While it seems many of the package managers that have switched to SAT solvers operate in a similar way, I am basing this alloy project off of the apt-get installer typically found in Debian installations.

All the code is contained in dependency_hell.als.

2 Formalizing the Problem

Now, I will formalize the problem I am attempting to tackle. In this section, I will discuss the inputs, outputs, and constraints on the operations.

First, we will talk about the packages themselves. These are the relevant attributes of each package.

1. **Locked:** If the package is allowed to be deleted, upgraded, or otherwise altered.
2. **Dependencies:** The list of packages that must be installed for this package to work. All of these packages must be installed.
3. **Conflicts:** The list of packages that CANNOT be installed for this package to work. If even one of these packages are installed, this package will not work.

Now, we will discuss the input of the problem.

1. **The Universe of Packages:** Any package that is already installed on the user's system or may need to be installed to upgrade the package in question.

2. **The Current State of Packages on the System:** The set of packages installed on the users system. This is the system before the upgrade begins.

3. **The Target Package to be Upgraded:** The package we wish to upgrade. This is the package the manager creates an upgrade plan for.

The output is simply a step-by-step upgrade plan. Each step will be one of three operations.

1. **Installation:** Installing a package from the universe.

2. **Deletion:** Deleting a package from the user's system.

3. **Upgrade:** This is essentially deleting a package and installing a new version of that package in one step.

We enforce valid steps in the upgrade plan.

1. **One Operation Takes Place:** Exactly one valid operation (mentioned above) takes place.

2. **Respect Package Lock:** No operation takes place on a locked package.

Last, we also enforce that each step is a valid state. Namely,

1. **All Dependencies Installed:** For all packages installed on the system, all dependencies of each of those packages are installed on the system as well.

2. **No Conflicting Packages Installed:** For all packages installed on the system, no conflicting package is installed.

3. **No Duplicate Packages:** No package is installed twice. Namely, two versions of the same package are not allowed to be installed on the system at once.

3 Representing the Problem in Alloy: dependency_hell.als

We use two primary ways to represent this problem in Alloy. First, graphs will represent what packages are available, installed, depend on each other, and conflict with one another. Second, we use Alloy's States to model each step of the upgrade plan.

3.1 Representing the Dependency Graphs

We break our dependency graph into roughly two components. Versions and packages.

We begin constructing our graph by defining our vertices. Rather than representing packages, we have them represent "Versions" and call them accordingly. Each Version has two types of edges. A directed dependency edge and an undirected conflict edge. The dependency edge go from a package to the dependency. Since a version conflict breaks the whole system, these are undirected edges.

A Package contains a set of Versions. Each instance of the set of Versions of that package. For instance, consider a package Git which may have versions 1, 2, and 3. Then, the Package is Git and the set contained in the Package are {Git v1, Git v2, Git v3}.

I chose to separate the packages and the versions like this to create an easier implementation. The number of additional atoms from combining the two are minimal for the examples we are using.

Next, we define the Universe of packages. The universe contains all Packages and Versions installed on the system and the Packages and Versions that may be needed to upgrade the package. For example, even if Less package isn't on the system, but is needed to upgrade Git, Less will be in the Universe.

3.2 Representing the Upgrade Plan

We use Alloy's states to represents the upgrade plan. Each State has a target, which is the package we are attempting to update and a set of Vertices called installed. Installed creates a dependency graph of all the Versions installed on the users system. We use predicates to enforce if a installed dependency graph is valid (has no conflicting packages, all dependencies installed, etc.). We also use predicates to enforce that the transition between States are valid (that is the installed dependency graph is manipulated in valid ways).

4 Themes

Despite my best efforts, I was not able to create a comprehensible theme for this project. Instead, I opted to use the table feature when viewing an instance. In particular, I looked at the table with columns "this/State", "installed", and "target". It should be the sixth table.