

Integrating Domain-Specific Package Managers into Distribution Package Management Systems

Michael Homer
michael@gobolinux.org

Abstract

This paper describes a mechanism for incorporating the domain-specific package managers that have become increasingly common in some language communities into the distribution's package management system. The described system aims to embrace, rather than extinguish, these third-party systems.

1 Motivation

While package managers such as LuaRocks[1], RubyGems[2], CPAN[3], PEAR[4], Haskell's Hackage[5], and others are very convenient for authors, frequently to the point of being the only obvious means of distribution for some interpreted languages[6], they do not integrate with the operating system distribution's package manager at all. The individual components may all be packaged up as upstream packages, but the packages will inevitably be out-of-date and it is likely that many will remain unpackaged entirely. This is frustrating for users and leads them to seek alternative means of installing the software. This situation has been cited as the motivation for other efforts to solve this problem [7].

Often users have been tempted to install their own trees for these alien package systems - either built under `/usr/local` or worse, allowed to pollute the main tree and create conflicts with the system package manager. These conflicts can be destructive, and in either case there is no integration between the alien system and that provided by the distribution. They are agnostic

of each other's existence, and a distribution package requiring Ruby-GTK+ cannot have that dependency satisfied by a Ruby Gem. Users are forced to choose between their distribution manager and the third-party system, to the detriment of both.

As well, for some languages even when the software is installable without using the packaging system there is a database that must be updated in order for the language to find the installed libraries, such as for Haskell [8]. This is another complication for installing libraries for these languages, and can be fatal when trying to install packages locally if the distribution manager also manipulates the database. A whole parallel installation of the language is necessary, with even more complication involved than for the packaging system.

Users who migrate between distributions and operating systems, including Windows, Mac OS X, or *BSD may also prefer to use the familiar language-specific interface.

The Aliens system was developed to bridge this gap within GoboLinux, a distribution with an alternative filesystem hierarchy[9], but is not tied to any particular distribution. A third-party ("alien") package management system can be incorporated in order to be fully integrated with the distribution's package management, with each requiring only a wrapper using a defined generic protocol. This structure should be transferable to other distributions, and such transfer is encouraged.

2 Overview

Each alien package manager, such as LuaRocks, is given full control of a subtree in the system. In GoboLinux, this tree is `/System/Aliens/LuaRocks`. The user may manipulate this using the ordinary and familiar `luarocks` command, following instructions from the library author's website or elsewhere. The Aliens system remains out of the user's way here, and they need not even know it exists. They can install, remove, find, and update the alien packages in exactly the way they are used to.

A program within the system package manager may also depend on the presence of a Lua library available through LuaRocks, as part of its ordinary package dependencies. When the system package is installed and has a dependency of the form `LuaRocks:json >= 1.0` the Aliens system calls on the `luarocks` command to check whether the module is installed already and

install it if necessary. If the dependency is already met it will do nothing further and not try to upgrade the existing rock.

There is no overhead of repackaging or keeping the wrapped package up-to-date with new releases. There is also no possible duplication of packages between the distribution and the alien system.

The entire Aliens subsystem is independent of the overlying distribution package manager and should be transferable into any other system, provided only that a few hooks into dependency resolution and installation can be provided.

3 Rationale and Alternatives

It was clear to us that some improvement on the status quo was desirable, but the ultimate approach is somewhat heretical and bears explaining. There are other possible approaches to solving similar problems that were considered and rejected for our purposes.

Conceptually simplest is automated repackaging of the alien repositories into the distribution packaging format. This has been done before, for example for CPAN and Debian[7] and Arch Linux and Haskell's Hackage[10]. Some alien systems are more suitable for this than others; CPAN is particularly easy to repackage as it has a very thorough and well-defined format. GoboLinux had an existing tool for converting an individual CPAN package into its local format and one option was to extend this to automatically mirroring the entire tree. That solution does eliminate duplication and interference but does not satisfy all of the other motivations. The familiar command is not available and it is possible that some of its functionality will not be duplicated. There may also be licensing issues involved in this kind of wholesale reproduction of the repository [7], and there is a substantial infrastructure cost to it.

Another option is, as has been done by some distributions, to preach abstinence and declare that all methods of software installation but our distribution system and what we have packaged are unsupported and condemned. This is tempting and involves the least work on our part, but is unsatisfying and gives a poor user experience.

Also possible was to adopt a variant on endorsing the status quo, and attempting to mitigate the downsides as far as possible. A particular alternative location such as a directory under `/opt` or `/usr/local` could be endorsed

to avoid conflicts, and duplication of packages used from within the system could be accepted as necessary. This eventually evolved into the Aliens system as described here. As some of these systems seemed to see themselves as just a single piece of software as a whole, we could adopt that pose ourselves and tie in the subcomponents with special treatment rather than ignoring them. Integrating the alien manager with our system eliminated the drawbacks and gave us access to new functionality almost for free.

4 Implementation

Aliens uses a three-layered approach to implementation: the top-level integration with the distribution package manager, an intermediate Aliens layer to interface, and wrappers for each third-party system below. The structure is comparable to that of PackageKit, with the Aliens backend to the installation tool similar to some package manager's backend to PackageKit [11]. It focuses on the other side of the equation, below the distribution's package manager instead of above it.

4.1 Top-level integration

Integrating the system into a new package manager requires only one layer, implementing a couple of hooks into the dependency checking, adding support for specifying alien dependencies, and installation stages of the manager. For GoboLinux, these hooks were only a few lines of code each. They need only call out to the Aliens system and process the simple result.

4.2 Aliens layer

Next is the Aliens layer itself. This is what the top-level package manager speaks to, and what dispatches the requests to the relevant third-party system. It is generic across all the external systems, providing a common interface to querying and installation while allowing common functionality to be kept out of the wrappers themselves. It should be possible to use this layer and beyond on other distributions with minimal changes, though they might prefer to embed it natively into their system.

4.2.1 Interface

The layer is accessible both as a Python library and as an executable. The interface to the executable will be described here. All commands follow the form `Alien --mode AlienType:alienpkg [arguments ...]`. *AlienType* is the name of the alien system and of the wrapper implementing the interface for it.

<code>--met <i>min</i> [<i>max</i>]</code>	Query whether a version \in [<i>min</i> , <i>max</i>) is installed already. This is the bulk of dependency validation.
<code>--install [<i>version</i>]</code>	Trigger the alien system to install the package.
<code>--getversion</code>	Find the <i>currently-installed</i> version of the package.
<code>--getinstallversion [<i>min</i>] [<i>max</i>]</code>	Find the version that <i>would be</i> installed.

The command

```
Alien --met LuaRocks:json 1.0 2
```

will be true (exit 0) iff the “json” LuaRock is already installed, and its version is at least 1.0 but less than 2. Otherwise, it returns an undefined non-zero exit code for false. After that,

```
Alien --install LuaRocks:json
```

will install the package. In practice this command would not be called by the user directly, but only by the top-level packaging system internally. In GoboLinux, the executable is called only to install and the native embedding as a Python library for dependency resolution and all other functions, to allow for caching of results and clearer code. These are in fact the same fairly concise codebase.

4.3 Wrappers

Finally, the wrappers for the various external packaging systems. These are implemented as executables named after their system, and speak a common protocol with the Aliens layer. They can be implemented either as shell

scripts wrapping the command-line program or in the relevant language itself, making use of whatever libraries it has available for accessing the packaging system. This flexibility makes it easy to add new wrappers regardless of what facilities the language provides. They should be fully generic across distributions, translating between the defined interface and the wrapped packaging system.

These follow a similar interface to the Aliens layer, but with only the package name and not the name of the wrapper. The above commands would be translated to `Alien-LuaRocks --met json 1.0 2` and `Alien-LuaRocks --install json`. If the alien system has an unusual versioning system it can be used natively and processed by the wrapper, rather than rewriting them for the top-level system.

Wrappers currently exist for LuaRocks, RubyGems, and CPAN, and for the most part are fairly simple.

4.4 Configuration

Moving the trees of each alien system into a non-default location requires some reconfiguration of the wrapped system. Moving is recommended even for systems following the usual FHS to prevent installed files from clobbering those installed by the distribution package manager, and so that libraries installed for the language by other means (such as language bindings shipped with a library) are kept separately. For FHS-based systems these trees can go under `/opt`, which is reserved for “add-on application software packages” [12, 12], or `/var/lib`, which is for “state information pertaining to an application or the system” [12, 33].

This is more complicated for some systems than others: RubyGems, now included in Ruby itself, is a pathological case where changing the paths in use is almost impossible, while LuaRocks was relatively simple to reconfigure. The method of changing these locations will differ from system to system, but it is always recommended that the standard locations be left in place for searching where applicable.

5 Limitations

5.1 Reverse dependencies

Reverse dependencies - a dependency from an Alien package on a system package - are not addressed here. There are three possible solutions, all of which have some validity, and it is still an open question which would be best.

Firstly, how we have it in GoboLinux at the moment, and the simplest thing that will work. Where a dependency like this exists, we ensure that any packages in our system depending on the alien package also include the depended-upon package in their own direct dependencies, before the Alien package. This is simple and manageable, and scales well as new packages are introduced.

Another which we have actively considered is maintaining a dependency mapping within the Aliens system itself - remembering that “RubyGems:ruby-gtk” depends on “GTK+” in our system, and including that in our regular dependency resolution. This is slightly purer and reduces duplication of information, which may be particularly important if the minimum required version is updated or a new dependency is introduced. It would not be necessary to update all the existing packaging data using the Alien package. This also scales somewhat well, but introduces a new bottleneck into the process.

In many ways the best solution would be for the external package managers themselves to provide this information for each package. This is probably implausible, though it would be nice to have. Providing the information in a cross-distribution and cross-platform way would probably be an intractable problem, particularly given that some distributions habitually split upstream packages into hundreds of components in their systems and identifying the right one in each case borders on impossible. Package authors are unlikely to have the information to hand and may be using Windows or another platform.

LuaRocks already has a portion of this functionality available [13]:

A rock can specify an external package it depends on (for example, a C library), and give to LuaRocks hints on how to detect if it is present, typically as C header or library filenames. LuaRocks then looks for these files in a pre-configured search path and, if

found, assumes the dependency is fulfilled. If not found, an error message is reported and the user can then install the missing external dependency (using the tools provided by their operating system)

Extending this kind of behaviour to allow automatically satisfying the dependency and to include all the covered third-party systems would be the ideal solution. We are investigating enhancing it further with the LuaRocks community, but it looks unlikely to be perfect for the reasons above, and unlikely ever to cover all of the alien systems.

5.2 Rolling release

For obvious reasons the Aliens system is more suitable for distributions following at least a degree of a rolling release philosophy. Updates to the third-party repository may occur outside the control of the distribution, violating their closed fiefdom.

Even in this case there may be a degree of suitability. Many individual users of Debian follow the testing or unstable repositories, which have a higher degree of churn than the stable distribution. The existence of the Debian-CPAN mirror [7] demonstrates that at least some users of these distributions do want access to up-to-date packages from the external systems. It may also be possible to take a snapshot of the repositories and use these, but it would be inadvisable to restrict access to potential security updates like this.

6 Conclusion

The Aliens system aims to embrace the growing set of domain-specific package managers and incorporate them into the distribution's package management system. It enables a distribution package to depend on a package in the third-party system, and enables the user to use the familiar interface to this system without causing conflicts with packages from the distribution. It has been built in a distribution-agnostic manner as far as possible and should be portable to any POSIX-like system.

References

- [1] LuaRocks. <http://luarocks.org/>.
- [2] RubyGems. <http://docs.rubygems.org/>, <http://ruby-lang.org>.
- [3] CPAN: The Comprehensive Perl Archive Network. <http://cpan.org/>.
- [4] PEAR - PHP Extension and Application Repository. <http://pear.php.net/>.
- [5] Hackage. <http://hackage.haskell.org/packages/hackage.html>.
- [6] Debian/Ruby Extras - On RubyGems. <http://pkg-ruby-extras.aliioth.debian.org/rubygems.html>. “it is currently impossible to install most Ruby applications ... without using RubyGems”.
- [7] Jos Boumans. debian.pkgs.cpan.org – debified CPAN packages. <http://debian.pkgs.cpan.org/>.
- [8] Building and installing a package. <http://www.haskell.org/cabal/release/cabal-latest/doc/users-guide/builders.html#setup-register>.
- [9] Michael Homer, Hisham Muhammad, and Jonas Karlsson. An updated directory hierarchy for UNIX. In *linux.conf.au*, 2010.
- [10] Don Stewart. Automated package tracking for arch haskell. <http://archhaskell.wordpress.com/2009/08/15/automated-package-tracking-for-arch-haskell/>.
- [11] Richard Hughes. PackageKit - main page. <http://www.packagekit.org/>.
- [12] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. Filesystem Hierarchy Standard 2.3. Technical report, Filesystem Hierarchy Standard Group, 2004.
- [13] LuaRocks Wiki - Dependencies. <http://luarocks.org/en/Dependencies>. Accessed 2009-12-13.