# Building Linux Software with Conary

Michael K. Johnson *rpath, Inc.*
ols2005@rpath.com

## Abstract

This paper describes best practices in Conary packaging: writing recipes that take advantage of Conary features; avoiding redundancy with recipe inheritance and design; implementing release management using branches, shadows, labels, redirects, and flavors; and designing and writing dynamic tag handlers. It describes how Conary policy prevents common packaging errors. It provides examples from our rpath Linux distribution, illustrating the design principles of the Conary build process. It then describes the steps needed to create a new distribution based on the rpath Linux distribution, using the distributed branch and shadow features of Conary.

Conary is a distributed software management system for Linux distributions. Based on extensive experience developing Linux distributions and package management tools, it replaces traditional package management solutions (such as RPM and dpkg) with one designed to enable loose collaboration across the Internet. It enables sets of distributed and loosely connected repositories to define the components which are installed on a Linux system. Rather than having a full distribution come from a single vendor, it allows administrators and developers to branch a distribution, keeping the pieces which fit their environment while grabbing components from other repositories across the Internet.

If you do not have a basic working knowledge of Conary terminology and design, you may want to read the paper *Repository-Based System Management Using Conary,* in *Proceedings of the Linux Symposium, Volume Two, 2004*, kept updated at `http://www.rpath.com/technology/techoverview/`, which introduces Conary's design and vocabulary in greater detail. Terms called out in **boldface** in this paper without an explicit definition are defined in that overview.

## 1 Conary Source Management

Unlike legacy package management tools, Conary has integral management for source code and binaries, and the binaries are directly associated with the source code from which they have been built.

Conary stores source files in **source components**, and then uses a **recipe** (described later) to build **binary components** that it can install on a system. While most of the code that handles the two kinds of components is actually the same, the interface is different. The source components are managed using a Software Configuration Management (SCM) model, and the binary components are managed using a system management model.

The SCM model for managing source components is sufficiently familiar to experienced Concurrent Versioning System (cvs) or Subversion (svn) users; you can create a new source

component, check out an existing source component, add files to a source component, remove files from a source component (this is a single action in Conary, unlike cvs's two-step operation), rename files in a source component (like svn, but unlike cvs), and commit the current set of changes since the last commit (like svn, and like cvs except that the commit is atomic).

Conary has not been optimized as a complete SCM system. For example, we do not use it to manage subdirectories within a source component, and instead of importing source code into vendor branches, we import archives and apply patches to them. Conary may someday support a richer SCM model, since there are no significant structural or design barriers within Conary. It was a choice made for the sake of simplicity and convenience, and to focus attention on Conary as a system management tool rather than as an SCM tool—to encourage using Conary to track upstream development, rather than encourage using it to create forks.

In addition, Conary stores in the repository (without cluttering the directory in which the other files are stored) all files that are referenced by URL. When they are needed, they are downloaded (from the repository if they have been added to the repository; otherwise, via the URL) and stored in a separate directory. Then, when committing, Conary stores those automatically added source files (**auto-source files**) in the repository so that the exact same source is always available, enabling repeatable builds.

Like cvs, Conary can create branches to support divergent development (forks); unlike cvs, those branches can span repositories, and the repository being branched from is not modified, so the user only needs write privileges in the repository in which the branch is created. Unlike cvs (and any other SCM we are aware of), Conary also has two features that support converging source code bases: **shadows** that act like branches but support convergent instead of divergent development by providing intentional tracking semantics, and **redirects** that allow redirecting from the current head of the branch (or shadow) to any other branch (or shadow), including but not limited to that branch's parent. The redirect is not necessarily permanent; the branch can be revived. A redirect can even point to a different name package entirely, and so is useful when upstream names change, or when obsoleting one package in favor of another.

### 1.1 Cooking with Conary

Using a recipe to turn source into binary is called **cooking**.

The exact output produced by building source code into a binary is defined by several factors, among them the instruction set (or sets) that the compiler emits, and the set of features selected to be built. Conary encodes each combination of configuration and instruction set as a **flavor**. The configuration items can be system-wide or package-local. When cooking, Conary builds a **changeset** file that represents the entire contents of the cooked package.

There are three ways to cook:

- A **local** cook builds a changeset on the special `local@local:COOK` branch. It loads the recipe from the local filesystem, and can cook with recipes and sources that are not checked into the repository. It will download any automatic sources required to build.
- A **repository** cook builds a transient changeset on the same branch as the source component, and then commits it to

the repository. It loads the source component (including the recipe and all sources) from the repository, not the local filesystem. It finds all automatic sources in the repository. The same version can be built into the repository multiple times with different flavors, allowing users to receive the build that best matches their system flavor when they request a trove.

- An **emerge** builds a transient changeset on the special `local@local:EMERGE` branch, and then commits it to (that is, installs it on) the local system. Like a repository cook, it takes the recipe and all sources from the repository, not the filesystem. (This is the only kind of cook that Conary allows to be done as the root user.)

## 2 The Conary Recipe

All software built in Conary is controlled through a **recipe**, which is essentially a Python module with several characteristics. Here is an example recipe, which has a typical complexity level:[1]

```
class MyProgram(PackageRecipe):
  name = 'myprogram'
  version = '1.0'
  def setup(r):
    r.addArchive(
      'http://example.com/%(name)s-%(version)s.tar.gz')
    r.Configure()
    r.Make()
    r.MakeInstall()
```

The goal of Conary's recipe structure is not to make all packaging trivial, but to make it possible to write readable and maintainable complex recipes where necessary, while still keeping the great majority of recipes extremely simple—and above all, avoiding boilerplate that needs to be copied from recipe to recipe. This example is truly representative of the the most common class of recipes; the great majority of packaging tasks do not require any further knowledge of how recipes work. In other words, this example is representative, not simplistic. New packagers tend to find it easy to learn to write new Conary packages.

However, some programs are not designed for such easy packaging, and many packagers have become used to the extreme complexity required by some common packaging systems. This experience can lead to writing needlessly complex and thereby hard-to-maintain recipes. This, in turn, means that while reading the RPM spec file or Debian rules for building a package can be an easy way to find a resolution to a general packaging problem when you are writing a Conary recipe, trying to translate either of them word-by-word is likely to lead to a poor Conary package.

The internal structure of objects that underlie Conary recipes makes them scale gracefully from simple recipes (as in the example) to complex ones (the kernel recipe includes several independent Python classes that make it easier to manage the kernel configuration process). Some of the more complex recipe possibilities require a deeper structural understanding.

- The recipe module contains a class that is instantiated as the **recipe object** (`MyProgram` in the example above). This class declares, as class data, a `name` string that matches the name of the module, a `version` string, and a `setup()` method. This class must be a subclass of one of a small family of abstract superclasses (such as `PackageRecipe`).
- Conary calls the recipe object's `setup()` method, which populates lists of things to do; each to-do item is represented by

[1] No, I do not like two-column formatting for technical papers, either.

an object. There are **source objects**, which represent adding an archive, patch, or source file; and **build objects**, which represent actions to take while building and installing the software. Additionally, there are pre-existing lists of **policy objects** to which you can pass extra information telling them how to change from their default actions. The `setup()` function returns after preparing the lists, before any build actions take place.

- Conary then processes the lists of things to do; first all the source objects, then all the build objects, and finally all the policy objects.

It is important to keep in mind that unlike RPM spec files and portage ebuild scripts (processed in read-eval-print loop style by a shell process) or Debian rules (processed by make), a Conary recipe is processed in two passes (three, if you count Python compiling the source into bytecode), because it both constrains the actions you can or should take and makes Conary more powerful. For example, you should not add sources inside a Python conditional (instead, you unconditionally add them but can choose not to apply them based on a conditional), but this constraint allows Conary to always automatically store copies of all sources that it has fetched by URL instead of being explicitly committed locally.

Another important data structure in a recipe is the macros object, an enhanced dictionary object that is an implicit part of every recipe. Almost every string used by any of the different kinds of objects in the recipe—including the strings stored in the macros object itself—is automatically evaluated relative to the contents of the macros object, meaning that standard Python string substitution is done. Thus, you do not have to type `%r.macros` after every string; the substitution is done within the functions you call. It also means that macros can reference each other. Be aware that changes to the macros object all take place before any list processing. This means that an assignment or change to the macros object at the end of the recipe will affect the use of the macros object at the beginning of the recipe. This is an initially non-obvious result of the multi-pass recipe processing.

The string items contained in the macros object are colloquially referred to by the Python syntax for interpolating dictionary items into a string. Thus, `r.macros.foo` is usually referred to as `%(foo)s`, because that is the way you normally see it used in a recipe.

The macros object contains a lot of immediately useful information, including the build directory (`%(builddir)s`), the destination directory (`%(destdir)s`) that is the proxy for the root directory (`/`) when the software is installed, many system paths (`%(sysconfdir)s` for `/etc` and `%(bindir)s` for `/usr/bin`), program names (`%(cc)s`), and arguments (`%(cflags)s`).

## 3 Recipe Inheritance and Reference

Conary recipes can reference each other, which makes it easier to use them to create a coherent system.

When many packages are similar, it is easy to end up with boilerplate text that is copied between packages to make the result of cooking them reflect that similarity. That boilerplate can be encoded in `PackageRecipe` subclasses, stored in recipes that are normally never cooked because they function as abstract superclasses. The recipes containing

those abstract superclasses are loaded with the `loadSuperClass()` function, which loads the latest version of the specified recipe from the repository into the current module's namespace. The main class in the recipe then descends from that abstract superclass. (The inheritance is pure Python, so it is possible to use multiple inheritance if that is useful.) This mechanism serves two purposes: it reduces transcription errors in what would otherwise be boilerplate text, and it reduces the effort required to build similar packages. It also allows bug fixes that are generic to be made in the superclass and thus automatically apply to all the subclasses.

Sometimes, you want to reference a recipe without inheriting from it. In that case, you use a similar function called `loadInstalled()`, which loads a recipe while preferring the version that is installed on your system, if any version is installed on your system. (Otherwise, it acts just like `loadSuperClass()`.) For example, you can load the perl recipe in order to programmatically determine the version of perl included in the distribution, without actually requiring that perl even be installed on the system.

## 4 Dynamic Tag Handlers

Conary takes a radically different approach to install-time scripts than legacy package management tools do. Typical install-time scripts are package-oriented instead of file-oriented, primarily composed of boilerplate, and often clash with rollback (for those package management tools that even try to provide rollback functionality). Conary tags individual files, instead; this file is a shared library, that file is an init script, another file is an X font. Then, at install time, once for each transaction (which may be many troves all together), it calls **tag handler** scripts to do whatever is required with the tagged files.

Bugs in tag handlers demonstrate some of the good characteristics of this system. When a bug in a tag handler is fixed, that bug is fixed for all packages in one place, without any need to copy code or data around. It is fixed even for older versions of packages built before the tag handler was fixed (as long as the package in question is not the one that implements the tag handler). Also, tag handlers can be called based not only on changes to the tagged files, but also changes to the tag handler itself, including bug fixes.

While tag handlers are clearly an improvement over legacy install-time scripts, it is still possible to make many of the same mistakes.

**Overuse** It is always slower to run a script than to package the results of running a script. Therefore, if you can perform the action at packaging time instead, do so. For example, put files in the `/etc/cron.d/` directory instead of calling the `crontab` program or editing the `/etc/crontab` file.

**Excessive dependencies** The more programs a script calls, the more complex the dependency set needed to run it. Circular dependencies are worse; they will break, one way or another. Finally, calling more programs increases the risk of accidental circular dependencies.

**Inappropriate changes** Modifying file data or metadata that is under package management and storing backup copies of files are generally inappropriate for any package manager, not just Conary. With Conary, though, a few more things are inappropriate, including adding users and groups to the system (it is too late; the files have already been created).

**Poor error handling** Scripts that do not check for error return codes can easily wreak havoc by assuming that previous actions succeeded. (We have discovered that if you are having difficulty managing error handling in a tag handler, you may be taking an approach that is needlessly complex. Look for a simpler way to solve the problem.)

It is also easy to misapply assumptions developed while writing legacy install-time scripts to tag handler scripts. The most important thing to remember is that everything in Conary, including tag handlers, is driven by changes. Therefore, if a package includes five files with the `foo` tag, and none of those tagged files is affected by an update, the `foo` tag handler will simply not be called. If only two of the five files is modified in the update, the `foo` tag handler will be called, but asked to operate only on the two modified files.

Write tag handlers with rollbacks in mind. This means that if the user does the inverse operation in Conary, the effect of the tag handler should be inverted as well. Most post-installation tasks merely involve updating caches, and often the list of affected files is not even required in order to update the cache. These cases are easy; just run the program which regenerates the cache.

When inventing new tag names, keep the tag mechanism in mind. `mypackage-script` is a horrible name for a tag handler, because it initiates or perpetuates the wrong idea about what it is and how it works. The name of a tag handler should describe the files so tagged. The sentence "*file* is a(n) *tag name*" should sound sensible, as in "`/lib/libc-2.3.2.so` is a `shlib`" or "`/usr/share/info/gawk.info.gz` is an `info-file`". Following this rule carefully has helped produce clean, simple, fast, relatively bug-free tag handlers.

# 5 Policy

After unpacking sources, building binaries, and installing into the `%(destdir)s`, Conary invokes an extensive set of policy objects to normalize file names, contents, locations, and semantics, and to enforce inter-package and intra-package consistency.

Policy objects are invoked serially and have access to the `%(destdir)s` as well as the `%(builddir)s`. Early policy objects can modify the `%(destdir)s`. After all `%(destdir)s` modification is finished, Conary creates a set of objects that represents the set of files in the destdir. Later policy then has that information available and can modify the packaging, including marking configuration files, marking tagged files, setting up dependencies, and setting file ownership and permissions.

The `policy.Policy` abstract superclass implements much of the mechanism that policies need. Many policies can simply list some regular expressions that specify (positively and negatively) the files to which they apply by default, and then implement a single method which is called for each matching file. The superclass provides a rich generic exception mechanism (also based on regular expressions) and a file tree walker that honors all the regular expressions. Policies are not required to use that superstructure; they can implement their own actions wherever necessary.

Policies can take additional information, as well as exceptions. For example, policy divides the files in a package into components automatically, but when the automatic assignment makes a bad choice, you can pass additional information to the `ComponentSpec` policy to change its behavior. Whenever you want to create multiple packages from a single recipe,

you have to give the `PackageSpec` policy information on which files belong to what package. When you use non-root users and groups, you need to provide user information using the `User` policy and group information using the `Group` policy.

This object-oriented approach is fundamentally different from RPM policy scripts. In contrast to Conary policy, RPM policy scripts are shell scripts (or are driven by shell scripts), and have an all-or-nothing model. If they do not do exactly what you want, you have to disable them and do by hand every action that the scripts would have done. This means that if you do not keep up with changes in the RPM policy scripts, your re-implementation may not keep up with changes in RPM. Also, because this restriction is onerous, RPM policy scripts cannot be very strict or very useful; they have to be limited in power and scope to the least common denominator. By contrast, the rich exception mechanism in Conary policy allows the policy to be quite strict by default, allowing for explicit exceptions where appropriate. This allows Conary to enable a rich array of tests that enable packaging quality assurance, with the tests run before any cooked troves are committed to the repository or even placed in a changeset.

Policies have access to the recipe object, mainly for the macros and a cache of content type identification that works like an object-oriented version of the `file` program's magic database and is therefore called the **magic cache**. This makes it easy to predicate policy action on ELF files, ar files, gzip files, bzip2 files, and so forth. The magic objects (one per file) sometimes contain information extracted from the files, such as ELF sonames and compression levels in gzip and bzip2 files.

Some policy, such as packaging policy, is intended to remain a part of the Conary program per se. However, many policies will eventually be defined (or expanded) outside of Conary, by the distribution, via **pluggable policy**. This will be a set of modules loaded from the filesystem, probably with one policy object per module, and a system to ensure that ordering constraints are honored.

### 5.0.1 Policy Examples

One of the best ways to describe what policy can do is to describe some examples of what it does do. As of this writing, there are 58 policy modules, so these few examples are by no means exhaustive; they are merely illustrative.

The `FilesInMandir` policy is very simple, and demonstrates how one line of code and several lines of data can implement effective policy. It looks only in `%(mandir)s`, `%(x11prefix)s/man`, and `%(krbprefix)s/man`, does not recurse through subdirectories, and within those directories only considers file entries, not subdirectory entries. Any recipe that needs to actually store a file in one of those directories can run `r.FilesInMandir(exceptions='%(mandir)s/somefile')` to cause the policy to ignore that file. All of this action specified so far requires only three simple data elements to be initialized as class data in the policy. Then a single-line method reports an error if the policy applies to any files, automatically ignoring any exceptions that have been applied from the recipe.

This simple policy effectively catches a common disagreement about what the `--mandir` configure argument or the `MANDIR` make variable specifies; the autotools de-facto standard is `/usr/share/man/` but some upstream packages set it instead to be a subdirectory thereof, such as `/usr/share/man/man1`, which would cause all the man pages to go in the wrong place by default. Having this policy

to catch errors makes it feasible for Conary to set `--mandir` and `MANDIR` by default, and fix up the exceptional cases when they occur.

The `RemoveNonPackageFiles` policy modifies the `%(destdir)s` and is even simpler in implementation than `FilesInMandir`. It lists as class data a set of regular expressions defining files that (by default) should not be included in the package, such as .cvsignore files, .orig files, libtool .la files, and so forth. It then has a single-line method that removes whatever file it is applied to. Again like `FilesInMandir`, a simple exception overrides the defaults; a recipe for a package that actually requires the libtool .la files (there are a few) can avoid having them removed by calling `RemoveNonPackageFiles(exceptions=r'\.la$')`—note the leading `r`, which tells Python that this is a "raw" string and that it should not interpret any `\` characters in the string.

Both `RemoveNonPackageFiles` and `FilesInMandir` use the built-in directory walking capabilities of the `Policy` object. Most policy does, but it is not required. The `NormalizeManPages` policy is different. It implements its own walking over each man directory (the same ones that the `FilesInMandir` policy looks at), and removes any accidental references to `%(destdir)s` in the man pages (a common mistake), makes maximal use of symlinks, makes sure that all man pages are compressed with maximal compression, and then makes sure that all symlinks point to the compressed man pages. It uses almost none of the built-in policy mechanism; it merely asks to be called at the right time.

The `NormalizeCompression` policy ignores man pages and info pages, since they have their compression normalized intrinsically via other policy. It automatically includes all files that end in `.gz` and `.bz2`. Then, for each file, it looks in the magic cache (which is self-priming; if no entry exists, it will create one), and if it really is a gzip or bzip2 file and is not maximally compressed, it recompresses the file with maximal compression.

Finally, the `DanglingSymlinks` policy uses packaging information, looking at which component each file is assigned to. It is not enough to test whether all symlinks in a package resolve; it is also important to know whether a symlink resolves to a different component, since components can be installed separately. There are also special symlinks that are allowed to point outside the package, including conshelper symlinks (which create an automatic requirement for the `usermode:runtime` component) and symlinks into the /proc filesystem. The `DanglingSymlinks` policy simply warns about symlinks that point from one component into another component built from the same package (except for shared libraries, where `:devel` components are expected to provide symlinks that cross components); symlinks that are not resolved within the package and are not explicitly provided as exceptions cause the cook to fail.

## 6 Recipe Writing Best Practices

Disclaimer: Not all recipes written by rpath follow these best practices. We learned many of these best practices by making mistakes, and have not systematically cleaned up every recipe. So the first rule is probably not to worry about mistakes; Conary is pretty forgiving as a packaging system, and we have tried to make it fix mistakes and warn about mistakes. You absolutely do not need to memorize this list to be a Conary packager. It's quite possible that the majority of new Conary recipes are fewer than ten lines of code. Relax, everything is going to be all right!

The best practices are somewhat arbitrarily divided into general packaging policy suggestions, conventions affecting building sources into binaries, and conventions affecting the `%(destdir)s`.

### 6.1 General Packaging Policy

Before worrying about packaging details, start working on consistency at a higher level. Names of packages, structure of recipes, and versions are best kept consistent within a repository, and between repositories.

#### 6.1.1 Simple Python Modules

Starting simple: recipes are Python modules. Follow Python standards as a general rule. In particular, do not use any tabs for indentation. Tabs work fine, but you will be running the `cvc diff` command many times, and tabs make diff output look a little odd because the indentation levels are not all even, and when you mix leading tabs and leading spaces, the output looks even weirder.

Second, follow Conary standard practice. Conary standard practice has one significant difference from Python standard practice: the self-referential object is called `r` (for recipe) instead of `self` because it is used on practically every line.

To get the most benefit from Conary, write your recipes to make it easy to maintain a unified patch that modifies them. That way, someone who wants to shadow your recipe to make (and maintain) a few small changes will not be stymied. The most basic way to do this is to keep your recipe as simple as possible. Don't do anything unnecessary. Don't do work that you can count on policy to do for you, such as recompressing gzip or bzip2 files with maximal compression turned on, or moving files from `/etc/rc.d/init.d` to `%(initdir)s`, unless policy can't do the job quite the right way—and in that case, add a comment explaining why, so that the person shadowing your recipe does not walk into a trap.

Not doing lots of make-work has another important benefit. The less you do in the recipe, the less likely you are to clash with future additions to policy, and the more likely you are to benefit from those additions. Policy will continue to grow to solve packaging problems as we continue to find ways to reduce packaging problems to general cases that have solutions which we can reliably (partially or completely) automate.

#### 6.1.2 Follow Upstream

Whenever possible, follow upstream conventions. Use the upstream name, favoring lower case if upstream sometimes capitalizes and sometimes does not, and converting "`-`" to "_" because "`-`" is reserved as a separator character. Do not add version numbers to the name; use version numbers in the name only if the upstream project indisputably uses the version number in the name of the project. Conary can handle multiple versions simultaneously just fine by using branches; no need to introduce numbers into the name. The branches can be created quite arbitrarily; they do not need to be in strict version order. Just avoid clashing with the label used for release stages by choosing a very package-specific tag. Example tags rpath has used so far are `sqlite2`, `gnome14`, and `cyrus-sasl1`. The head of `conary.rpath.com@rpl:devel` for sqlite is sqlite version 3, and the branch `conary.rpath.com@rpl:devel/2.8.15-1/sqlite2/` contains sqlite version 2, as of this writing version 2.8.16, giving a full version string

of `/conary.rpath.com@rpl:devel/2.8.15-1/sqlite2/2.8.16-1`

When possible and reasonable, one upstream package should produce one Conary package. Sometimes, usually to manage dependencies (such as splitting a text-mode version of a program from a graphical version so that the text-mode version can be installed on a system without graphical libraries installed) or installed features (such as splitting client from server programs), it is reasonable to have one upstream package produce more than one Conary package. Rarely, it is appropriate for two upstream packages to be combined; this is generally true only when the build instructions for a single package require multiple archives to be combined to complete the build, and all the archive files really are notionally the same project; they aren't just a set of dependencies.

If you have to convert "`-`" to "`_`" in the name, the following convention may be helpful:

```
r.macros.ver = \
  r.version.replace('_', '-')
r.mainDir('%(name)s-%(ver)s')
```

### 6.1.3 Redirects

Finally, if the upstream name changes, change the name of the package as well. This means creating a new package with the new name, and then changing the old package into a redirect that points to the new package. Users who update the package using the old name will automatically be updated to the new package.

Alternatively, if you change the package that provides the same functionality, you can do the exact same thing; from Conary's point of view there is no difference. For example, rpath Linux used to use the old mailx program to provide `/usr/bin/mail`, but switched to the newer nail program for that task. The mailx recipe was then changed to create a redirect to the newer nail package. Anyone updating the mailx package automatically got the nail package instead.

A redirect is not necessarily forever. The old recipe for mailx could be restored and a new version cooked; the nail recipe could even be changed to a redirect back to mailx. If that happened, an update from the older version of mailx would completely ignore the temporary appearance of the redirect to nail. (Not that this is likely to happen—it just would not cause a problem if it did.)

Redirects pointing from the old troves to the new troves solve the "obsoletes wars" that show up with RPM packages. In the RPM universe, two packages can each say that they obsolete each other. In the Conary world, because redirects point to specific versions and never just to a branch or shadow, this disagreement is not possible; the path to a real update always terminates, and terminates meaningfully with a real update. There are no dead ends or loops.

## 6.2 Build Conventions

Compiling software using consistently similar practices helps make developers more productive (because they do not have to waste time figuring out unnecessarily different and unnecessarily complex code) and Conary more useful (by enabling some of its more hidden capabilities).

### 6.2.1 Use Built-in Capabilities

Use build functions other than `r.Run` whenever possible, especially when modifying the `%(destdir)s`.

Build functions that do not have to start a shell are faster than `r.Run`. Using more specific functions enables more error checking; for example, `r.Replace` not only is faster than `r.Run('sed -i -e ...')` but also defaults to raising an exception if it cannot find any work to do. These functions can also remove build requirements (for example, for `sed: runtime`), which can make bootstrapping simpler and potentially faster.

Most build functions have enough context to prepend `%(destdir)s` to absolute paths (paths starting with the `/` character) but in `r.Run` you have to explicitly provide `%(destdir)s` whenever it is needed. For example, `r.Replace('foo', 'bar', '/baz')` is essentially equivalent (except for error checking) to `r.Run("sed -i -e 's/foo/bar/g' %(destdir)s/baz")` in function, but the `r.Replace` is easier to read at a glance.

Many build functions automatically make use of Conary configuration, including macros. `r.Make` automatically enables parallel make (unless parallel make has been disabled for that recipe with `r.disableParallelMake`), and automatically provides many standard variables.

A few build functions can actually check for missing `buildRequires` items. For example, if you install desktop files into the `/usr/share/applications/` directory using the `r.Desktopfile` build command, it will ensure that `desktop-file-utils:runtime` is in your `buildRequires` list, and cause the build to fail if it is not there.

In particular, `r.Configure`, `r.Make`, and `r.MakeParallelSubdirs` provide options and environment variables that the autotools suite, and before that, the default make environment, have made into de-facto standards, including names for directories, tools, and options to pass to tools. Consistency isn't just aesthetic here; it also enhances functionality. It enables `:debuginfo` components that include debugging information, source code referenced by debugging information, and build logs. It allows consistently rebuilding the entire operating system with different compiler optimizations or even different compilers entirely. This is useful for testing compilers as well as customizing distributions.

### 6.2.2 Macros

Use macros extensively. In general, macros allow recipes to be used in different contexts, allow changes to be made in one place instead of all through a recipe, and can make the recipe easier to read.

Using macros for filenames means that a single recipe can be evaluated differently in different contexts. If you refer to the directory where initscripts go as `%(initdir)s`, the same recipe will work on any distribution built with Conary, whether it uses `/etc/rc.d/init.d/` or `/etc/init.d/`.

Using macros such as `%(cc)s` for program names (done implicitly with make variables when calling the `r.Make*` build actions) means that the recipe will adapt to using different tools, whether that is for building an experimental distribution with a new compiler, or for using a cross-compiler to build for another platform, or any other similar purpose.

Use the macros that define standard arguments to pass to programs, such as `%(cflags)s`, and modify them intelligently. Instead of overwriting them, just modify them, like `r.macros.cflags += ' -fPIC'` or `r.macros.cflags = r.macros.cflags.replace('-O2', '-Os')`

or `r.macros.dbgflags = r.macros.dbgflags.replace('-g', '-ggdb')`

Creating your own macros for text that you would otherwise have to repeat throughout your recipe makes the recipe more readable, less susceptible to bugs from transcribing existing errors and making errors in transcription, and easier to modify. You might, for example, do things like `r.macros.prgdir = '%(datadir)s/%(name)'`

Creating your own macros can also help you make your recipes fit in 80 columns, for easier reading in the majority of terminal sessions.

```
r.macros.url = 'http://reallyLongURL/'
r.addArchive('%(url)s/%(name)s-%(version)s.tar.bz2')
```

### 6.2.3 Flavored Configuration

When configuring software (generally speaking, before building it, but it is also possible for configuration to control what gets installed rather than what is built), make sure that the configuration choices are represented in the flavor. When a configuration item depends on the standard set of **Use flags** for your distribution, use those. If there is no system-wide Use flag that matches that configuration item, you can create a **local flag** instead.

A lot of configuration is encoded in the arguments to `r.Configure`. We commonly use the variable `extraConfig` to hold those. There are two reasonable idioms:

```
extraConfig = ''
if Use.foo:
  extraConfig += ' --foo'
r.Configure(extraConfig)
```

and

```
extraConfig = []
if Use.foo:
  extraConfig.append('--foo')
r.Configure(
  ' '.join(extraConfig))
```

In either case, referencing `Use.foo` will cause the system-wide Use flag named "foo" to be part of the package's flavor, with the value that is set when the recipe is cooked.

If you need to create a local flag, you do it with the package metadata (like `name`, `version`, and `buildRequires`):

```
class Asdf(PackageRecipe):
  name = 'asdf'
  version = '1.0'
  Flags.blah = True
  Flags.bar = False
  def setup(r):
    if Flags.blah:
      ...
```

Now the `asdf` package will have a flavor that references `asdf.blah` and `asdf.bar`. The values provided as metadata are defaults that are overridden (if desired) when cooking the recipe.

## 6.3 Destdir Conventions

Choosing how to install files into `%(destdir)s` can determine how resilient your recipe is to changes in Conary and in upstream packaging, and how useful the finished package is.

### 6.3.1 Makefile Installation

Using `r.Make('install')` would not work very well, because it would normally cause the Makefile to try to install the software directly

onto the system, and you would soon see the install fail because of permission errors. Instead, use `r.MakeInstall()`. It works if the Makefile defines one variable which gives a "root" into which to install, which by default is called `DESTDIR` (thus the `%(destdir)s` name). If that does not work, read the Makefile to see if it uses another make variable (common names are `BUILDROOT` and `RPM_BUILD_DIR`), and pass that in with the `rootVar` keyword: `r.MakeInstall(rootVar='BUILDROOT')`

Sometimes there is no single variable name you can use. In these cases, there is a pretty powerful "shotgun" available: `r.MakePathsInstall`. It re-defines all the common autotools-derived path names to have the `%(destdir)s` prepended. This works for most of the cases without an install root variable. Sometimes you will find (generally from a permission error, less commonly from reviewing the changeset) that you need to pass an additional option: `r.MakePathsInstall('WEIRDDIR=%(destdir)s/path/to/weird/dir')`

For a few packages, there is no Makefile, just a few files that you are expected to copy into place manually. Use `r.Install`, which knows when to prepend `%(destdir)s` to a path, and knows that source files with any executable bit set should default to mode 0755 when packaged, and source files without any executable bit set should default to 0644; specify other modes like `mode=0600`—do not forget the leading 0 that makes the number octal. (Conary does look for mode values that are nonsensical modes and look like you left the 0 off, and warns you about them, but try not to depend on it; the testing is heuristic and not exhaustive.) Like other build actions, `r.Install` will create any necessary directories automatically. If you want to install a file into a directory, make sure to include a trailing `/` character on the directory name so that Install knows that it is intended to be a directory, not a file. (It is this requirement that allows it to make directories automatically.)

### 6.3.2 Multi-lib Friendliness

Conary does its best to make all packages multi-lib aware. Practically all 32-bit x86 libraries are available to work on the 64-bit x86_64 platform as well, making Conary-based distributions for x86_64 capable of running practically any 32-bit x86 application, not just a restricted set that uses some of the most commonly-used libraries.

The first thing to do is to always use `%(libdir)s` and `%(essentiallibdir)s` instead of `/usr/lib` and `/lib`, respectively. Furthermore, for any path that is not in one of those locations, but still has a directory named "lib" in it, you should use `%(lib)s` instead of `lib`. Conveniently, for programs that use the autotools suite, Conary does this for you, but when you are reduced to choosing directory names or installing files by hand, follow this rule.

On 64-bit platforms on which `%(lib)s` resolves to `lib64`, Conary tries to notice library files that are in the wrong place, and will even move 64-bit libraries where they belong, while warning that this should be done in the packaging rather than as a fixup, because there is probably other work that also needs to be done. Conary warns about errors that it cannot fix up, and causes the cook to fail.

Conary specifically ensures that `:python` and `:perl` components are multi-lib friendly, since there are special semantics here; some `:python` or `:perl` packages have only interpreted files and so should be found in the 32-bit library directory even on 64-bit platforms; others have libraries as well, and should be in

the 64-bit library on 64-bit platforms. Putting the 64-bit object files in one hierarchy and interpreted files in another hierarchy would create path collisions between 32-bit and 64-bit `:python` or `:perl` components.

### 6.3.3 Direct to Destdir

Occasionally, an upstream project will include a package of data files that is intended to be unpacked directly into the filesystem. Instead of unpacking it into the build directory with `r.addSource` and then copying it to `%(destdir)s` with `r.Copy` or `r.Install`, use the `dir` argument to `r.addArchive`. Normally, `dir` is specified with a relative path and thus is relative to the build directory, but an absolute path is relative to `%(destdir)s`. So something like `r.addArchive('http://example.com/foo.tar.bz2', dir='%(datadir)s/%(name)s/')` will do what you want in just one line. Not only does it make for a shorter recipe with less potentially changing text to cause shadow merges to require manual conflict resolution, it is also faster to do.

### 6.3.4 Absolute Symlinks

Most packaging rules tell you to use relative symlinks (`../...`) instead of absolute (`/...`) symlinks, because it allows the filesystem to continue to be consistent even when the root of the filesystem is mounted as a subdirectory rather than as the system root directory; for example, in some "rescue disk" situations.

This rationale is great, but Conary does something even better. It automatically converts all absolute symlinks not just to relative symlinks, but to *minimal* relative symlinks. That is, if you create the absolute symlink `/usr/bin/foo -> /usr/bin/bar`, Conary will change that to `/usr/bin/foo -> bar`, and `/usr/bin/foo -> /usr/lib/foo/bin/foo` to `/usr/bin/foo -> ../lib/foo/bin/foo`. Therefore, for Conary, it is best to use absolute symlinks in your `%(destdir)s` and let Conary change them to minimal relative symlinks for you.

## 6.4 Requirements

Conary has a very strong dependency system, but it is a bit different from legacy dependency systems. The biggest difference is that depending on versions is very different from any other packaging system. Because Conary (by design) does not have a function that tries to guess which upstream versions might be newer than another upstream version, you cannot have a dependency that looks like "upstream version 1.2.3 or greater."

Because Conary has the capability for a rich branching structure, trying to do version comparisons even on Conary versions for the purposes of satisfying dependencies fails utility tests. If you say that a shadow does not satisfy a dependency that its parent satisfies, then shadows are almost useless for creating derivative distributions. However, if you say that a shadow does satisfy a dependency that its parent satisfies, then a shadow that intentionally removes some particular capability relative to its parent will falsely satisfy versioned dependencies. Trying to do strict linear comparisons in the Conary version tree universe just does not work.

Conary separates dependencies into different spaces that are provided with individual semantics. Each ELF shared library provides a **soname** dependency that includes the ABI (for example, `SysV`), class (`ELFCLASS32` or `ELFCLASS64` encoded as `ELF32` or `ELF64`, respectively), and instruction set (`x86`, `x86_64`,

ppc, and so forth) as well as any symbol versions (GLIBC_2.0, GLIBC_2.1, ACL_1.0 and so forth). The elements are stored as separate **flags.** Programs that link to the shared libraries have a dependency with the same format. These dependencies (requirements or provisions) are coded explicitly as a soname dependency class. The order in which the flags are mentioned is irrelevant.

Trove dependencies are limited to components, since they are the only normal troves that directly reference the files needed to satisfy the dependencies. (Filesets also contain files, but they are always files pulled from troves, so they are not the primary sources of the files, and they are not intended for this use.) By default, a trove dependency is just the name of the trove, but it can also include **capability flags**, whose names are arbitrary and not interpreted by Conary except checking for equality (just like upstream versions).

This provides the solution to the version comparison problem. Trove A's recipe does not really require upstream version 1.2.3 or greater of trove B:devel in order to build. Instead, it requires some certain functionality in trove B:devel. The solution, therefore, is for package B to provide a relevant capability flag describing the necessary interface, and for trove A's recipe to require trove B:devel with that capability flag. The capability flag could be as simple as 1.2.3, meaning that it supports all the interfaces supported by upstream version 1.2.3 (the meaning of any package's capability flag is relative to only that package). So package B's recipe would have to call r.ComponentProvides('1.2.3') and trove A's recipe would have to require 'B:devel(1.2.3)'.

This solution does require cooperation between the packagers of A and B, but only in respect to a single context. This means that you may choose to shadow trove B in order to add this capability flag in the context of your derived distribution, if your upstream distribution does not provide the capability your package requires.

Do not add trove capability flags without good reason, especially for build requirements. They add complexity that is not always useful. Usually, the development branch for a distribution just needs to be internally consistent, and adding lots of capability flags will just make it harder for someone else to make a derivative work from your distribution, particularly if they are deriving from multiple distributions at once (a reasonable thing to do in the Conary context).

### 6.4.1 Build Requirements

Conary's build requirements are intentionally limited to trove requirements.

In general, there are two main kinds of build requirements: :runtime components (and their dependencies) for programs that need to run at build time, and :devel components (and their dependencies) for libraries to which you need to link.

Build requirements need to be added to a list that is part of recipe metadata. Along with name and version, there is a list called buildRequires, which is simply a list of trove names (including, if necessary, flavors, branch names, and capability flags, but not versions). It can be extended conditionally based on flavors.

```
buildRequires = [
  'a:devel(A_CAPABILITY)',
  'gawk:runtime',
  'pam:devel[!bootstrap]',
  'sqlite:devel=:sqlite2',
]
```

```
if Use.gtk:
  buildRequires.append(
    'gtk:devel')
```

The `buildRequires` list does not have to be exhaustive; you can depend on transitive install-time dependency closure for the troves you list. That is to say, in the example above, you do not have to explicitly list `glib:devel`, because `gtk:devel` has an install-time requirement for `glib:devel`. (The `buildRequires` lists do not themselves have transitive closure, as that would be meaningless; you never require a `:source` component in a `buildRequires` list, and the dependencies that the other components carry are install-time dependencies.)

Build requirements for `:runtime` components can be a little bit hard to find if you already have a complete build environment, because some of them are deeply embedded in scripts. It is possible to populate a changeroot environment with only those packages listed in the `buildRequires` list and their dependencies, then chroot to that environment and build in it and look for failures, but it is not likely to be a very useful exercise. The best approach here is to add items to address known failure cases.

Build requirements for `:devel` components are much simpler. Cook the recipe to a local changeset, and then use `conary showcs --deps foo-1.0.ccs` to show the dependencies. (Better yet, use `--all` instead of `--deps` and review the sanity of the entire changeset.) Then, for each soname requirement listed under each `Requires` section, add the associated component to the list. (Right now, this takes too many steps; you need to look for the library, then use `conary q --path /path/to/library` to find the name of the component. In the future, there will be a simple command for looking these up, and we are considering automating the whole process of resolving soname requirements to `buildRequires` list entries.)

### 6.4.2 Runtime Requirements

The best news about runtime requirements is that you can almost ignore the whole problem. The automatic soname dependencies handle almost everything for you without manual intervention.

There are also some automatic file dependencies, which present a little bit of an asymmetry. Script files automatically require their interpreters. That is, if a file starts with `#! /bin/bash` that file (and thereby its component) automatically has a requirement for `file: /bin/bash` added to it. However, there is no automatic provision of file paths. This is because files are not primarily accessed by their paths, but rather by a long numeric identifier (rather like an inode number in a filesystem, but much longer, and random rather than sequential in nature). Files can be tagged as providing their path, but this must be done manually. In practice, this is not a big problem; most programs that normally act as script interpreters are already tagged as providing their paths, and so the exceptions tend to exist within a single trove. Those cases are easy to fix; Conary refuses to install a trove saying that it cannot resolve a `file: /usr/bin/foo` dependency, but the trove itself contains the /usr/bin/foo file. Just add `r.Provides('file', '%(bindir)s/foo')` to the recipe.

The hard job with any dependency system is working out the dependencies for shell scripts. It is not practical to make shell dependencies automatic for a variety of reasons (including the fact that shell scripts could generate additional dependencies from the text of their in-

put), and so it remains a manual process. If you are lucky, the package maintainer has listed the requirements explicitly in an INSTALL or README file. If not, you need to glance through shell scripts looking for programs that they call. Since this is not a new problem, you can in practice (for some packages, at least), find the results of other people's efforts in this direction by reading RPM spec files and dpkg `debian/rules`. This also tends to be an area where dependencies accrete as a result of bug reports.

There is one place where you need to be much more careful about listing the requirements of shell scripts: you must explicitly list all the requirements of the tag handlers you write. This should not be a great burden; most tag handlers are short and call only a few programs. But if you do not list them, Conary cannot ensure that the tag handlers can always be run, which can jeopardize not only successful installation but also rollback reliability.

## 7 Release Management

Building software into a repository is already an improvement over legacy package management, but release and deployment need more management and process than just building software into a versioned repository. Several of Conary's features are useful for managing release and deployment; groups, branches, shadows, redirects, and labels can all help.

Different release goals or deployment needs will result in different policies and processes. This paper uses some concrete examples to demonstrate how Conary features can support a release management process, but the mechanisms are flexible and can support diverse processes. A release can go through one QA step or ten separate QA steps without changing the fundamental processes. Release management and deployment have many of the same needs, so this paper will refer generally to release management except when it is useful to distinguish between the two.

The capabilities needed for release management include:

**Staging** Collecting troves (including locally modified versions) to create a coherent set for promotion to the next step in the process.

**Access Control** Mandatory or advisory controls on who can or should access a set of troves.

**Maintenance** Controlled updates for sets of troves.

In addition, the jargon for talking about Linux distributions is somewhat vague and used in conflicting ways. The following definitions apply to this discussion.

**Distribution** A notionally-connected set of products consisting of an operating system and related components. A distribution might last for years, going through many major release cycles. Examples include rpath Linux, Foresight Linux, Red Hat Linux, Fedora Core, Debian, Mandrake (now Mandriva) Linux, CentOS, cAos, and Gentoo.

**Version** One instance of a distribution product, encompassing the entire "release cycle," which might include steps like alpha, beta, release candidate, and general availability. Examples include rpath Linux 1, Red Hat Linux 7.3, Fedora Core 2, etc.

**Stage** A working space dedicated to a task.

**Release** An instance of any step in the distribution release management process. (This is a slightly unusual meaning for "release;" "version" and "release" are often used almost interchangeably, but for the purposes of this discussion, we need to differentiate these two meanings.) This might be alpha 1 release candidate 1, alpha 1 release candidate 2, alpha 1, beta 1, beta 2, release candidate 1, general availability, and each individual maintenance update.

A release of a version of a distribution is defined (in Conary terms) by a unique version of an inclusive group that defines the contents of the distribution. In rpath Linux, that group is called `group-os`.

### 7.1 Example Release Management Process

The policy and much of the process in this example is synthetic, but the version tree structure it demonstrates (including the names for labels in the example) is essentially the one that we have defined for rpath Linux.

The development branch called `/conary.rpath.com@rpl:devel` (hence, `:devel`) is where the latest upstream versions are committed. At some point, a group defining a distribution is shadowed to create a **base** stage, `/conary.rpath.com@rpl:devel//rel-base` (hence, `//rel-base`) allowing unfettered development to continue on the `:devel` development branch, while controlled development (a state sometimes called "slush," by analogy from "freeze") is now possible on `//rel-base`.

Given a very simple, informal release management process—say, one where only one person is doing all the work, following all the process from the time that the initial release stage is created, and in which maintenance does not need to be staged—this single shadow creating a single stage might be sufficient. However, in order to allow any controlled development to happen in parallel with the full release engineering process, and in order to allow maintenance work to be staged, a two-level stage structure is necessary.

Therefore, when the controlled development has reached the point where an alpha release is appropriate, another shadow is created on which to freeze that release. This allows controlled development to continue on the release base stage: `/conary.rpath.com@rpl:devel//rel-base//rel-alpha` (hence, `//rel-alpha`). Build the shadowed `group-os` (or whatever you have called your inclusive group), and the version you have just created is a candidate alpha release. Cycle through your test, fix, rebuild process until you have a version of `group-os` that meets your criteria for release as an alpha. At this point, that specific version of `group-os`, say `group-os=/conary.rpath.com@rpl:devel//rel-base//rel-alpha/1.0-1-2.3.1-35`, is your alpha 1 release.

Note that during the test, fix, rebuild process for alpha 1, development work aimed at alpha 2 can already be progressing on the base stage. Fixes that need to be put on the alpha stage for alpha 1 can either be committed to the base stage and thence shadowed to the alpha stage, or if further development has happened on the base stage that could destabilize the alpha stage, or the immediate fix is a workaround or hack and the right fix has not yet been committed to the base stage, the fix, workaround, or hack can be committed directly to the alpha stage.

Then for the alpha 2 cycle, you re-shadow everything from the base stage to the alpha stage, and start the test, fix, rebuild process over again. When you get to betas, you just create a beta stage: `/conary.rpath.com@rpl:`

devel//rel-base//rel-beta (hence, //rel-beta) and work with it exactly as you worked with the alpha stage. Finally, when you are ready to prepare release candidates, build them onto the final release stage /conary.rpath.com@rpl:devel//rel-base//rel (hence, //rel) in the same way.

Note that it is possible to do all your release staging from first alpha to ongoing maintenance onto the release stage //rel. However, using separate named stages for alpha, beta, and general availability can be a useful tool for communicating expectations to users. It is your choice from a communications standpoint; it is not a technical decision.

During maintenance, do all of your maintenance candidates on the base stage, and promote the candidate inclusive group to the release stage by shadowing it when all the components have passed all necessary tests.

All the stages are really **labels**, as well as shadows. You can shadow any branch you need to onto the base stage, and you will probably want to shadow troves from several branches. Not just /conary.rpath.com@rpl:devel but also branches like /conary.rpath.com@rpl:devel/1.5.28-1-0/cyrus-sasl1/ (hence, :cyrus-sasl1) for different versions where both versions should be installed at once. With that :cyrus-sasl1 branch and cyrus-sasl 2 from the :devel branch both shadowed onto the base stage and thence to the release stages, the command conary update cyrus-sasl will put both versions on your system.

When the release stage is no longer maintained, you might choose to cook redirects (perhaps only for your inclusive group, perhaps for all the packages) to another, still-maintained release. This is purely a matter of distribution policy.

## 8 Derived Distributions

The possibilities for creating derived distributions are immense, but a few simple examples can show some of the power of creating derived distributions.

The simplest example of a derived distribution is the 100% derived distribution. If you merely want control over deployment of an existing distribution, just treat the parent distribution's release stage (//rel) as your base stage, and create a shadow of it in your own repository. You will end up with something like: /conary.rpath.com@rpl:devel//rel-base//rel//conary.example.com@rpl:myrel (hence, //myrel). Then, whenever a group on //rel passes your acceptance tests, you shadow it onto //myrel.

If you are doing anything more complicated, you may want to set up two stages; your own base stage and your own release stage. If you are doing this, you probably do not want to shadow a release stage as your base stage; you will end up with very long version numbers like 1.2.0-2.0.4.0-1.0.1.0; each shadow adds a "." character with a trailing number. You probably want either to shadow the parent distribution's base stage, or even create your own base stage. To create your own base stage, create your own shadow of the parent distribution's inclusive group and make your own changes to it. Those changes might be adding references to some unique troves from your own repository, or to shadows in your repository from other repositories.

You could create a private corporate distribution, with your repository inaccessible from

outside, that contains your internally developed software, or third-party proprietary software to which you have sufficient license. (A source trove doesn't necessarily have to contain source files; it could contain an archive of binary files which are installed into the `%(destdir)s`.) You could create a distribution in which everything is identical to the parent, except that you have your own kernel with special patches that support hardware you use locally that is not yet integrated into the standard kernel, and it has two extra packages which provide user-space control for that hardware.

It is also possible to make significant changes. For example, Foresight Linux[2] is built from the same development branch as rpath Linux, but about 20% of its troves are either specific to Foresight Linux or are shadows that are changed in some way in order to meet Foresight's different goals as a distribution; rpath Linux is meant to be very "vanilla," with few patches relative to upstream packages and therefore easy to branch from, while Foresight Linux is intended to provide the latest innovations in GNOME desktop technology and optimize the rest of the distribution to support this role.

## Conclusion

Conary combines system management and software configuration management, sharing features between the two models and implementing them both using a distributed repository that combines source code and the binaries built from that source code. It brings a unique set of features that simplify and unify system management, software configuration management, and release management. This new model drastically reduces the cost and complexity of creating customized Linux distributions. The best practices discussed in this paper help you take advantage of this new paradigm most effectively.

[2] `http://www.foresightlinux.com/`

# Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

## Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*