Conaryopedia

2012.02.01

Conaryopedia: 2012.02.01

rPath welcomes feedback on this and other documentation. Report issues in the Documentation project at the rPath Issue Tracking System (issues.rpath.com).

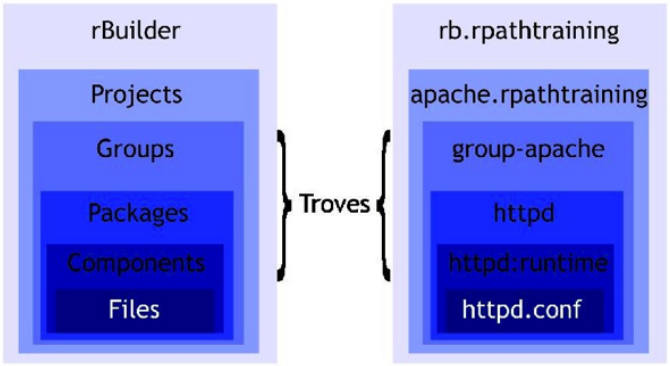
Introduction to Conaryopedia

"Conaryopedia" is the project name for moving Conary documentation to docs.rpath.com. This includes moving many resources from the rPath Wiki (*wiki.rpath.com*), bringing information up-to-date, and adding new information to the mix. It is rare that Conaryopedia will be referenced as a complete work; the goal is to produce individual chapters as resources linked from docs.rpath.com/conary.

1. Conary Concepts

Before you tackle packaging software with Conary, read this chapter to familiarize yourself with some Conary concepts. The sections are presented in a carefully selected order to maximize initial understanding.

Figure 1.1. Conary concepts



1.1 Repositories, Repository Hostnames, and Troves

A Conary *repository* is a network-accessible software repository at the heart of Conary's version control features. Conary runs as a service rather than as a data store like APT and YUM repositories. Repositories are important to these types of users in a Conary-based environment:

- The *system administrator* uses commands on an existing Conary-based system to find software and retrieve updates from one or more repositories.
- The *appliance assembler* picks items out of one or more repositories and uses those items to assemble a complete appliance. This is usually handled within the rPath Cloud Engine user interface, rPath's main tool for creating and maintaining complete Conary-based systems.
- The *packager* checks out repository contents and checks in (or commits) new and modified contents, including software that can be installed and managed with Conary.

Each repository has an identity in Conary called a *repository hostname*. The repository hostname is not a DNS-resolvable hostname, but is an identifier that looks similar to a DNS hostname which Conary maps to a repository URL. This mapping is called a repository map (*repositoryMap*) in Conary configuration. A Conary-based system can have any number of repository maps to use when searching for software. The following is an example repository hostname and its corresponding Conary configuration:

```
demoapp.example.com
```

```
repositoryMap demoapp.example.com https://rbuilder.example.com/repos/demoapp
```

A *trove* is an addressable unit in Conary, either in a repository or on a Conary-based system. By "addressable unit," we mean that you can identify a single unique trove by its name, version string, and flavor (the last two are defined in later sections). Two troves can have the same name but be different versions, or they can be the same name and

version but different flavors. This allows you to address older versions or different flavors of the same software from the same repository.

You will see the word "trove" in log files and user interface messages where it serves as a more generic term for referring to packages, components, and groups (each defined in later sections).

1.2 Labels

You can identify where content exists in a Conary repository by using its *label*. A label has the following format:

```
__________@_____:_______
```

The three parts of a label are:

- *repository hostname* - The part to the left of the @.
- *namespace* - Between the @ and the :.
- *tag* - To the left of the colon.

The namespace and the tag together make up a *branch name* within that repository.

Together, the two sides of the "@" in the label convey "what repository" and "what branch within that repository."

```
repository_hostname@branch_name
```

Conary separates the branch name into two parts using a colon. Those two parts are the *namespace* and the *tag*. If you have a standalone Conary repository, you could invent any namespace or tag you want to use. Conary will see any two things as being on the same "label" as long as those labels match.

```
repository_hostname@namespace:tag
```

rPath's rPath Cloud Engine gives meaning to the namespace and tag to better organize appliance software in its Conary repositories. In rPath Cloud Engine, the namespace is typically a common name or abbreviation for your organization, and the tag is a hyphen-separated string identifying the appliance and where the trove is in the appliance's development cycle. rPath Cloud Engine creates and maintains these branch names (namespaces and tags) automatically based on product definitions it maintains for the appliance.

```
demoapp.example.com@corp:demoapp-1-devel
```

Refer to troves in a repository as being "on" a particular label. This conveys both which trove you're talking about and where to find it.

For example, the following statement is a way of saying "the *demoapp:runtime* trove is on the label *demoapp.example.com@corp:demoapp-1-devel*":

```
demoapp:runtime=demoapp.example.com@corp:demoapp-1-devel
```

1.3 Packages and Components

In Conary package management, a *package* is a trove that represents all the files associated with a single application or customization. The way you work with Conary packages varies based on your role:

- As a *system administrator*, you'll install and manage packages that contain the software the system needs. Conary configuration helps you identify one or more labels where the system can find and download both packages and ongoing package updates. From there, you'll use the Conary commands for querying, adding, updating, and removing packages on each system.
- As a *packager*, you'll create a Conary package for software applications and make packages available in one or more Conary repositories. Use rPath Cloud Engine, rBuild, and rMake to create Conary packages so they can manage the packaging environment in a way that Conary alone cannot.

In Conary, a package consists of one or more components. A *component* contains the files that serve a particular functional role in the application. Conary automatically assigns files to components when the package is created. Components are named with the package name plus the component name, separated by a colon (:), as in *example:lib*. For Linux software, Conary can identify each file's role easily and predictably when the software developers follow the *Linux Filesystem Hierarchy Standard*. *Table 1.1. Component Assignment Examples* shows some examples of how components line up with the Linux filesystem hierarchy:

Table 1.1. Component Assignment Examples

Packaged File or Directory	Target Directory	Component Name
example.exe	*/usr/bin/*	*example:runtime*
libexample.so	*/usr/lib/*	*example:lib*
example/*	*/usr/share/doc/*	*example:doc*

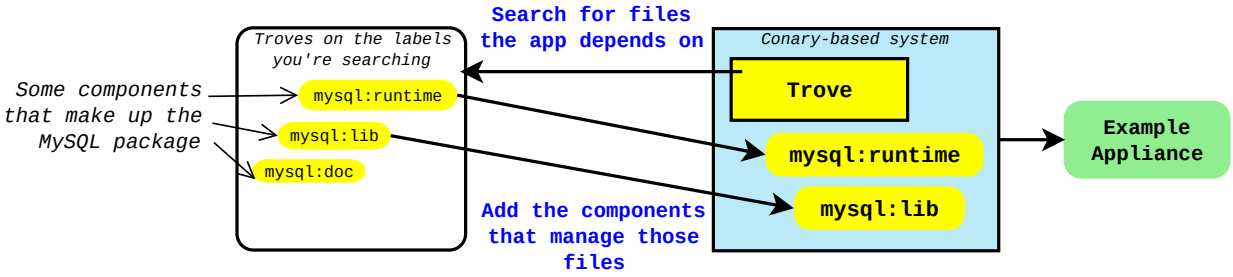
The role of a component in Conary is to provide efficient dependency resolution when software is installed. A *dependency* is extra software that another application needs in order to function properly. For example, a wiki application that's programmed in PHP requires PHP software, a Web server for HTTP access, and possibly database software to hold its data. *Dependency resolution* refers to successfully finding and installing those dependencies.

As a *system administrator*, you'll see Conary attempt to find and resolve dependencies. Conary identifies dependencies at the file level and resolves dependencies at the component level. This means that Conary installs only the components it needs to provide a given file from a package instead of trying to install the entire package. The following sequence describes the dependency resolution process when installing a trove on a Conary-based system:

1. When you try to install a trove, Conary identifies the trove's dependencies at the file level. This means that Conary looks for just the files it needs to make the package work.
2. For each file it needs, Conary searches the labels that the system is configured to search, looking for the first trove it can find that manages that file.
3. If Conary finds all the files it needs, it reports the components it will need to install in order to add those files to the system. As a system administrator, you can then tell Conary to install those components along with the trove you're targeting.

   If Conary doesn't find all the files it needs, it reports that failure with the information about the files it couldn't find. To correct the failure, you'll need to expand your search to other labels as described in the later section on Conary commands and configuration.

Figure 1.2. Dependencies are determined at the file level, resolved at the component level



Though component-level dependency resolution keeps the system from having software it doesn't need, it also means that Conary does not allow any two troves to manage the same file on the same Conary-based system. As a *packager*, if you want to control a file that's already managed by another component, best practice is to override that component's package with your own customizations. This is covered in later sections about shadowing and deriving packages.

1.4 Source Components, Recipes, and Builds

A *source component* in Conary is a trove you'll find in a Conary repository that's used to build one or more troves. The source component is named by appending ":source" to the trove name (such as *example:source* for the *example* package). A source component contains the instructions for how to build the trove (package or group), and it might also contain other files such as the original application. The role of the source component in Conary is similar to that of a source RPM (SPRM) in RPM package management.

As a *system administrator* you'll probably only encounter a source component when you're browsing repositories. You may see a source component as one of the components associated with a package or group. However, source components cannot be installed on a Conary-based system.

As a *packager*, you'll check out and work with source components from Conary repositories. The contents of the source component for a package is also called the *package source*. Just as most software programs require source code that's written in a programming language like Java or Python, Conary packages require package source with at least one file that's written in the Conary application programming interface (API). That file is called a *recipe*.

The *recipe* file in the package source is the set of instructions used to assemble a package. A recipe is similar to, but much shorter than, a specification file for an RPM. As a *packager*, editing the recipe is probably where you'll spend the most time.

Conary packaging tools use your package source to *build* the package. The result of building the package is called a *package build*, and that's what you can actually install on a Conary-based system. Both the package source and the package build can be checked in to the repository, and this happens automatically when building packages in the rBuilder user interface.

> ### Package, Package Source, and Package Build
>
> Because the word "package" alone might refer to the source, build, or both, this guide will use "package source" and "package build" to help distinguish between them when necessary.

1.5 Changesets and Revisions

For software programmers using a version control system, there is a procedure for checking out code and checking it in again, sometimes called committing changes to the code. Each new check-in creates a new version of that code that's tracked in the version control system. If you want to see the changes in the code between any two versions, you can use the version control system to view those differences.

Troves in a Conary repository can be compared to determine the differences between versions. As a *packager*, you might want to view the differences between two package sources. As a *system administrator*, you might want to compare differences between two package builds before updating a system from one build to another.

Before you can compare two troves, you need to know the *revision* of each trove you're comparing. The revision is assigned to a trove when a packager checks in or commits that trove to a Conary repository. The revision is made up of three values: an upstream version, a source count, and a build count. These values are separated by hyphens, such as in *1-2-1* and *3.4a-3-2*. The upstream version is the string of characters assigned to the *version* variable in the package recipe, and this is covered more in the sections on editing recipes. The other two values are the tracking numbers for the revisions themselves:

- The *source count* is the revision of the package source. In other words, each time a packager commits changes to files in the source component (without changing the upstream version), Conary increments the source count.

  Example trove revisions with consecutive source counts: 16e.7-**1**-2, 16e.7-**2**-1, 16e.7-**3**-1

- The *build count* is the revision of the package build. In other words, each time a packager commits a new build of the same package source, Conary increments the build count.

  Example trove revisions with consecutive build counts: 16e.7-3-**1**, 16e.7-3-**2**, 16e.7-3-**3**

As previously stated, Conary assigns a revision value to a trove when it's committed to a Conary repository. The following are the rules Conary uses to assign a revision value to a trove:

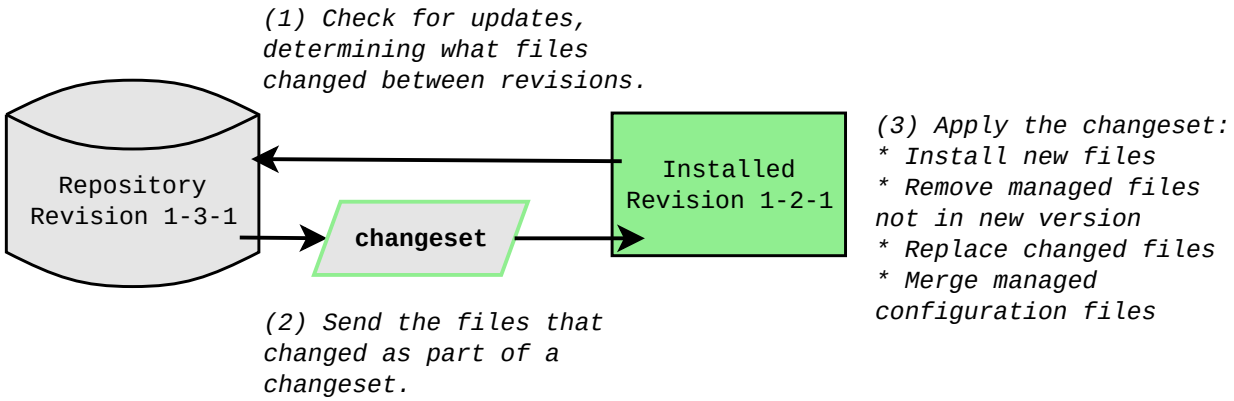
- If you change the upstream version within the recipe, the source and build counts both reset to 1. (Example: *2.4-3-1* might become *2.5-1-1*)
- For the same upstream version within the recipe, if you change the package source in any way, the source count is incremented, and build count resets to 1. (Example: *2.5-1-3* would become *2.5-2-1*)
- For the same upstream version within the recipe, if you create a new package build without changing the package source, the source count remains the same, and the build count is incremented. (Example: *2.5-2-1* would become *2.5-2-2*)
- Because a source component is not part of a package build, it does not have a build count in its revision. (Example: *2.5-2*)

When you know the revisions of two troves, you can compare the differences between any two revisions. Conary uses this difference comparison to create changesets used in certain operations. A *changeset* is a file containing a record of differences between one trove and other.

As a *system administrator*, changesets make updates more efficient. During an update, Conary calculates the difference between the package revision on the system and the package revision that you're updating to from the repository. Conary then creates a changeset from those differences, and it downloads and "applies" that changeset to update the

package. This means that you're only downloading the changes you need to apply, not the entire new revision of the package. Conary continuously tracks which revision of each package and other troves it currently has installed.

Figure 1.3. A changeset contains the files that need to change on the target system



Conary makes a special exception when a changeset impacts certain files on a Linux system:

- Conary-managed configuration files are merged so the system configuration is preserved for that specific deployment.
- Unmanaged files, which were not installed and maintained by Conary, are not affected during updates.

As a *packager*, note that a package build can be represented as an *absolute changeset*, which is a record of all the files that would be installed in order to install the package. You can see references to this changeset in the interface messages and log files associated with building the package. A later section describes how to use this changeset for testing the package build even before you commit to your repository.

1.6 Groups

In Conary, a *group* is a type of trove whose only purpose is to install and manage other troves. Conary recognizes a group by its naming convention: the *group-* prefix in its name.

As a *packager*, you can create groups to install and manage a set of troves (packages, components, or even other groups). This is especially useful to lock in specific versions of those troves that you know will work together when they're installed.

As a *system administrator*, note how groups restrict how some software can be updated. When you install or update a group, Conary will only update the software for a trove in the group if the group itself includes a newer revision of that trove. If you force updating any single trove to a newer revision, use caution: you might break compatibility between the troves in the group.

A group is created by writing a *group recipe*, which is covered in a later section. Like a package, a group has a source component, and this document will distinguish between the *group source* and *group build*.

An *appliance group* in rPath Cloud Engine is a special type of Conary group. The appliance group defines all the troves needed to build a complete appliance in rPath Cloud Engine. The *appliance group build* is one large trove that

can be used in two ways: (1) to create images used to deploy new systems, and (2) to indicate to existing deployments what that system revision should look like. Thanks to the appliance group build, a system administrator can use a single command to create, download, and apply a changeset that will bring the entire system up-to-date.

1.7 Version Strings

Each trove is considered unique when it has a unique combination of the following three things: a name, a version string, and a flavor. This section covers version strings, and the next section covers flavors.

A *version string* is the complete string of characters that indicates the label and revision of the trove along with any essential branching history from other labels. The syntax of the version string is a slash (/), a label, another slash (/), and the revision, as shown in the following syntax example:

```
/label/revision
```

The following is an example of a version string for a trove that originated on the *demoapp.example.com@corp:demoapp-4-devel* label:

```
/demoapp.example.com@corp:demoapp-4-devel/4.2-2-1
```

If the trove was branched from one label to another, the parent label is added to the beginning of the sequence, and all labels are separated by double slashes, as shown in the following syntax example:

```
/parentlabel//label/revision
```

The following is an example of a version string that originated on the *centos.rpath.com@rpath:centos-5* label and was branched to the *demoapp.example.com@corp:demoapp-4* label:

```
/centos.rpath.com@rpath:centos-5//demoapp.example.com@corp:demoapp-4-devel/2.6.1-3.1-1
```

If the trove is branched multiple times, Conary tracks all the parent labels, such as in the following example:

```
/centos.rpath.com@rpath:centos-5//demoapp.example.com@corp:demoapp-4-devel//
demoapp.example.com@corp:demoapp-4/2.6.1-3.1-1
```

To save space, Conary often abbreviates version strings by trimming off repository hostnames or namespaces that are the same. For example, the previous version string may be abbreviated by removing the *demoapp.example.com* repository hostname and the *corp* namespace in the last label:

```
/centos.rpath.com@rpath:centos-5//demoapp.example.com@corp:demoapp-4-devel//
demoapp-4/2.6.1-3.1-1
```

Another concept within the version string is the *branch*. In Conary, a branch is basically the entire part of the version string from the beginning slash through the last label. Remove the last slash and the revision from the version string to get the branch, as in the following example:

```
/demoapp.example.com@corp:demoapp-1-devel
/centos.rpath.com@rpath:centos-5//demoapp.example.com@corp:demoapp-4-devel
/centos.rpath.com@rpath:centos-5//demoapp.example.com@corp:demoapp-4-devel//demoapp-4
```

If you're packaging software for Conary, there are two ways to create a new branch for a trove:

- Create the original trove yourself and commit it to a given label.

- Derive or shadow an existing trove from another label. Both deriving and shadowing create a new development branch which remembers its parent trove and allows you to merge changes from that parent when desired.

1.8 Flavors

The same revision of a trove can be built simultaneously for multiple deployment conditions using *flavor specifications*. Each separate flavor specification results in a separate build, and a separate trove when committed to the repository. Example flavor specifications are "is: x86" for a 32-bit Intel 8086 processor instruction set and "vmware" for a VMware virtual machine. Except for instruction sets, the state of each specification is written with the operators shown in *Table 1.2. Operators for reading flavor specifications*.

Table 1.2. Operators for reading flavor specifications

Operators	Example	How it reads/how it's used
(no preceding mark)	vmware	*vmware* -- (building and identifying troves) the trove is exclusively built for systems running in VMware; (searching for troves) only return the trove if the system querying for it is *vmware*
!	!vmware	*not vmware* -- (building and identifying troves) the trove is exclusively built for systems not running in VMware; (searching for troves) only return the trove if the system querying for it is *!vmware*
~	~vmware	*prefers vmware* -- (searching for troves) in searching for troves from a system that is *!vmware*, return this trove only if there isn't already a trove that is *!vmware* or *~!vmware*
~!	~!vmware	*prefers not vmware* -- (searching for troves) in searching for troves from a system that is *vmware*, return this trove only if there isn't already a trove that is *vmware* or *~vmware*

Multiple flavor specifications together with their operators indicate the *flavor* of that trove. A flavor is written in square brackets ("[ ]") as a comma-separated list, except that the instruction set flavors are listed at the end, without a comma, starting with "is:" and separated by spaces. The following is an example of a flavor:

```
[!dom0, ~!domU, ~vmware, ~!xen is: x86 x86_64]
```

The way you use flavors will vary depending on your role:

- As a *system administrator*, the flavor of the system determines what you can install on the system. For example, your system cannot install a package with the *!domU* flavor specification if it has the *domU* flavor specification. Fortunately, Conary takes care of these determinations automatically, ignoring any repository contents that you cannot use when you're searching from the target system.

- As a packager, your tools should be configured to build packages for the deployment conditions you need. rPath Cloud Engine and rBuild handle this automatically based on your image definitions in the user interface. If you decide to use rMake or Conary packaging tools alone, you must provide the appropriate flavor specifications to represent the target deployment environment. If you need the package to install or function differently depending on that environment, you'll edit the recipe to include those conditions as covered in a later section.

2. Conary System Management

Conary has a series of commands for managing software on systems that are based on Conary package management, such as systems built with rPath's rBuilder. This chapter includes descriptions and examples of some of the most common `conary` commands, plus instructions for installing and using the *conary(1)* manual page which documents the `conary` commands in detail.

> **Root required for some commands**
>
> All commands that change software on the system require you to be either logged in as *root*, or using the `sudo` utility in front of the command. This is shown in the examples here using the **#>** prompt.

2.1 Common Conary Commands

Conary requires configuration that maps its internal features to the system and network resources it's interacting with.

Table 2.1. Get Conary configuration information

`conary config`	Display Conary all configuration details in the current scope.
`conary config | grep installLabelPath`	Display the installLabelPath setting, which is the sequence of labels that the system will search by default when it looks for software from your current scope.

Conary can provide a lot of useful information about the software installed on the system, and the software available for installing from Conary repositories, such as a platform repository in rBuilder. To reference what some of the information means, see "Conary Versions and Flavors" chapter at *docs.rpath.com/conary*.

Table 2.2. Get information about installed and available software

`conary query`	Query installed software for information about its components and update path. `$> conary query example` `$> conary q example --info` `$> conary q example --lsl` `$> conary q --path /etc/example.cfg`
`conary repquery`	Query software available for installation from Conary repositories. `$> conary requery \| more` `$> conary rq example --info` `$> conary rq --install-label rap.rpath.com@rpath:linux-2`
`conary verify`	Verify whether changes have been made to installed files.

Conary requires root access to update and roll back software on the system. For updating the entire system, Conary offers two approaches: *updateall* and *migrate*.

Table 2.3. Update and roll back software and the entire system

`conary update`	Install and update software from a Conary repository, including updating to an earlier version. `#> conary update example=rap.rpath.com@rpath:linux-2` `#> conary update group-example` `#> conary update changeset.css`
`conary updateall`	Update the system group and any additional components under those groups; *--apply-critical* installs only updates needed by Conary itself. `#> conary updateall` `#> conary updateall --resolve` `#> conary updateall --apply-critical` Use *--items* to list the items that Conary will update (without running the update): `#> conary updateall --items`
`conary migrate`	Migrate all Conary-managed components to a target version of a system group, including removing components that are not managed in the target version, and yet keeping configuration files and unmanaged files intact by default. Pass options to override some default behaviors with regards to removing unmanaged and managed files. `#> conary migrate group-toplevel-appliance` `#> conary migrate group-toplevel-appliance=example.rpath.org@corp:newbranch`
`conary remove`	Remove a file from the system and from Conary management, ensuring it is not reinstalled during Conary updates of the component that originally installed it. `#> conary remove /usr/share/example/example.doc`
`conary erase`	Uninstall software packages and components `#> conary erase example` `#> conary erase example:runtime`
`conary rollback`	Roll back a system to a prior state, before one or more update commands. `#> conary rollback 1` `#> conary rollback r.42`
`conary rblist`	List the update operations that would be reversed during each rollback. `#> conary rblist | more`

Conary pins kernels so that you can install multiple versions of the same *kernel* package. Linux administrators know this is essential to ensure you have at least one functional kernel on the system, even after an update. You can unpin old kernels if you want an update operation to overwrite/remove them. You can also pin or unpin other packages or components to control whether or not the system should manage multiple versions.

Table 2.4. Pinned items allow for installing multiple versions

`conary pin`	Pin an installed component or package to prevent it from being modified or removed during a `conary update` or `conary erase`. Kernel packages are pinned by default. `#> conary pin example` `#> conary pin kernel=2.6.17.11-1-0.1`
`conary unpin`	Unpin an installed component or package to allow it to be modified by a subsequent update or erase operation. `#> conary unpin example` `#> conary unpin kernel=2.6.17.11-1-0.1`

2.2 Install and Use the Conary Manual Page

If you're at the command line of a Conary-based system, the command `man conary` can be used to view the detailed manual page for the `conary` command. Some systems will not include software unless it is essential for supporting the applications. This might include the `man` utility as well as the `conary` documentation (component name *conary:doc*). Before installing these two new pieces, see the following two commands to install the man utility and the `conary` documentation, changing the rPath Linux label if necessary to match the label of the platform on which the system was built:

```
#> conary update man=conary.rpath.com@rpl:2 --resolve
#> conary update conary:doc=conary.rpath.com@rpl:2
```

The rPath Linux repository is one of the few repositories you'll access that should not need a repository map in Conary configuration. rPath has configured its Web servers to direct your requests to *conary.rpath.com* as appropriate.

3. Recipe Syntax and Structure

The bulk of time you spend packaging is likely to be writing and editing the package recipes. Each application you package has its own requirements to install, set up, and update the software. This chapter presents information you will need to develop and edit recipes.

You do not need to know how to program in any particular programming language to write a recipe. This chapter covers the syntax and conventions for your recipe.

Most common operations needed for packaging are already in pre-defined recipe actions. You can add custom actions in Python code to your recipe if necessary.

> ### Recipe Editors
>
> The rBuilder user interface includes recipe editors as part of the appliance contents editor, allowing you to modify your appliance group recipe and each of your package recipes within the rBuilder user interface. For more extensive package recipe editing, you can set up a command line environment such as the rPath Development Environment and use tools like rBuild instead of the rBuilder UI, and use your preferred Linux command-line text editor to edit recipe files.
>
> Once you being using a command line environment to edit recipes, DO NOT try to edit the same packages and groups in the rBuilder UI. This is because rBuilder will attempt to replace your custom work with its own code which was designed for UI-only packaging.

3.1 Recipe Syntax and Naming

Recipes are written in the domain-specific language of Conary, which itself is written in Python. Recipes should follow these Python syntax requirements and best practices:

- Use 4-space indention, not tabs
- Keep line lengths to 78 characters, exceeding this for strings without natural breaks
- Add comments in two ways: a single line starting with "#" or multiple lines with triple-double quotes (**"""**) on blank lines before and after the multiple lines

```
# This is a comment in the recipe
"""
This is a longer comment in the recipe.
Python programmers call this a docstring.
"""
```

- When breaking up an action line, indent following lines to the open parenthesis in the first line
- When breaking up a long argument over multiple lines, use a single quote (**'**) at the beginning and end of each line. For example:

```
r.Configure('options here'
            ' --more-option=here'
            ' --still-more-options')
```

The recipe file is named using the package name plus the `.recipe` extension at the end. For the *example* package, the recipe file is *example.recipe*.

Filter expressions are used in policy actions, and have the syntax of regular expressions with the following two special rules:

- A filter expression is always anchored at the beginning, meaning that it begins matching from the start of the string.
- A trailing slash (/) in a filter expression implies a **.*** immediately following it, meaning to match all paths under that directory.

For more information on policy actions, see *Policy Actions*.

3.2 Recipe Structure

Within the recipe file, you define a *class* used to build the package. The class name should be a *CamelCase* name that reflects the package name (though they don't have to match). For the *Example Application* package, the class name could be *Example*, *ExampleApp*, or something similar. This class name is included in the *class declaration* line which starts each recipe.

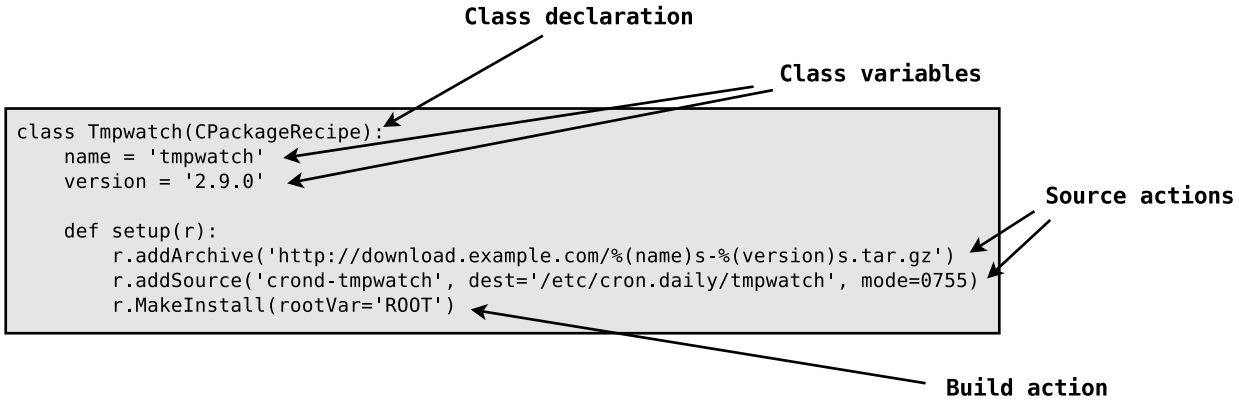
```
class ExampleApp(PackageRecipe):
```

Everything after the class declaration line is considered within that class and should be indented in **4-space increments**. The remaining lines in the recipe depend on whether or not you are using *factories*, a means of automating part of the recipe contents. This section looks at the basic structure of a Conary recipe without factories; the factory recipe structure is covered in *Chapter 4. Factories*.

The basic package recipe has the following possible parts after its class declaration, each covered in more detail in later parts of this document:

- Class variables -- used to describe various aspects of the package
- Processing actions -- used to direct building the package; includes source, build, and policy actions:
    - Source actions -- used to locate and obtain the files for creating the package
    - Build actions -- used to direct compiling software (if necessary) and installing software
    - Policy actions -- used to override Conary default behavior when packaging, such as removing empty directories or assigning files to certain components
- Use flags and flavors -- used to modify how the package is built for certain conditions

Figure 3.1. Package recipe for tmpwatch, starting from its C source code



The recipe must have at least two class variables in every recipe:

- the *name*, which must match the name of your package
- the *version*, which should reflect the version of the application software

The package version does not have to be a numeric value, but it cannot contain hyphens (-).

Your other package code goes within a single function you define in the recipe. This function is called *setup* or *unpack* (depending on your template). This function always passes the parameter *r*, and includes package recipe actions from *http://docs.rpath.com/conary/Conaryopedia/*.

Everything after the function definition line is considered within that function and should be indented an additional 4 spaces for a total of **8 spaces**.

No matter where they are within your recipe's method, actions are always performed in the following order:

- Source actions are performed first. All source actions in the recipe are performed in the order they are present in the recipe before any build or policy actions. For more information on source actions, see *Source Actions*.
- Build actions are performed second. All build actions in the recipe are performed after all source actions, but before any policy actions. Build actions are performed in the order they are present in the recipe. For more information on build actions, see *Build Actions*.
- Policy actions are performed after all source actions and all build actions in the recipe are performed. For more information on policy actions, see *Policy Actions*.

  Filter expressions in policy actions are applied in a *first match* order. For example, after a file's ownership is set by its matching a filter expression in an *r.Ownership* policy action, that file's ownership will not be affected by any matching *r.Ownership* filter expressions in the recipe.

Best practice is to write your recipe reflecting this order, and to test the recipe. Conary documentation for recipe actions indicates which actions are source, build, or policy actions.

Quick reference with `cvc explain`

To see the Conary API documentation for a particular recipe action while at a command prompt, use the `cvc explain` command followed by the recipe action, such as in the following example:

```
$> cvc explain addSource
```

Scroll through the documentation as needed to find the information you need, and then press "q" to quit.

4. Factories

A *factory* is a type of Conary recipe used to generate other recipes. Conary developers created factories so that tools like rBuilder's appliance content editor could automatically determine what kind of recipe was needed to package a given piece of software. Existing factories are tied to specific platforms provided by rPath, while the mechanics behind how a factory works is part of Conary package management.

> **Factories are used by Packagers, not System Administrators**
>
> Factories are probably invisible to you if you're just installing and maintaining Conary-based systems. Factories are an important feature if you're creating and maintaining packages for Conary-based systems.

To understand how factories work, first look at the object-oriented benefits Conary inherits from the Python programming language. For recipe code, this means you can chain together any code you need to build your recipe. This also means that any code you want to reuse in two or more packages (or groups) can be maintained in its own recipe class and called by the packages (or groups) that need it.

When you create a new package (or group) for Conary to use a factory, your new recipe automatically inherits all of the code from the factory recipe class. You can create a manifest file that the factory parses to determine where the application files reside. In many cases, this manifest file is all you'll need for your package; the *CONARY* state file has information that will do the rest when you build the package. If your target package needs some additional actions to override or supplement the factory actions, you can write a brief package recipe for that target package with just the code you need for those actions.

Working with factories differs when developing packages in rBuilder or developing packages outside of rBuilder. The following sections present information for these two situations.

4.1 Developing Packages in rBuilder

If you're developing packages for an appliance in rBuilder, rBuilder automatically assumes you want to use the *FactoryRecipeClass* for your selected platform. Factories will differ between platforms, even if they have the same name. rBuilder's content editor is designed to work with the appropriate factory recipe class based on what you're packaging. The following is an example of the code rBuilder might use for a package recipe.

```
# Factory-based recipe for ExamplePkg
class OverrideRecipe(FactoryRecipeClass):

    def preprocess(r):
        '''
        Place pre-processing actions here
        '''

    def postprocess(r):
        '''
        Place post-processing actions here
        '''
```

Unlike most package recipes that have a single method such as *setup()*, this *override recipe* structure includes two methods: *preprocess* and *postprocess*. Anything that's added in these methods will be added before or after the factory-generated recipe actions (respectively). This allows you to let the factory do all the work, or to add source, build, or policy actions to supplement the factory's code.

4.2 Developing Packages Outside rBuilder

If you're packaging outside of rBuilder, such as with rBuild or rMake, you can still take advantage of a platform's factory recipe classes. To do this, start by imitating the structure of the override recipe above.

The factory name is `factory-capsule-rpm` and is located in `group-factories` in your Centos, SLES, or RHEL platform repository.

> ### Filtering Through the Code
>
> Each *group-factories* maintained by rPath has additional Python code that you probably don't need to reference. Much of this code is designed for efficient operation through the rBuilder UI. For your reference as a packager outside of Package Creator, you should only use the non-Package-Creator factories listed in *Developing Packages Outside rBuilder*.

Instructions for using these factories are provided in an upcoming section. Conary documentation from http://docs.rpath.com/conary/ has additional reference for how to create and use your own custom factories.

5. Copying and Customizing Existing Conary Packages

You don't have to start from scratch just to customize a package for your own needs. Instead, you can copy packages and make the changes you want. You can even have Conary automatically keep track of changes upstream, allowing you to merge in those upstream developments at any time. This chapter covers copying and customizing existing packages to save yourself time and effort, and to ensure consistency across many different Conary-managed systems.

There are three approaches to modifying existing packages for your appliance:

- **Derive** -- Create a *derived* package when you want to "drop in" a minor change, like a configuration file or graphic.
- **Shadow** -- Create a *shadow* when you want to make major changes, like recompiling the application's source code with different options, or building the package with different requirements. Merge changes from the parent branch periodically to bring in upstream changes.
- **Clone** -- Create a *clone* when you want to make changes, and you don't want to merge changes from the parent branch over time.

Prior to diving in to this chapter, be sure you understand the version and label concepts defined in *Conary Versions and Flavors* from *docs.rpath.com/conary*.

5.1 Shadowing and Deriving

Both a shadow and a derived package start with a shadowing operation in Conary. If you're using rBuild, though, separate commands ensure you don't have to make manual changes that turn the shadow into a derived package.

But, how do you know whether to shadow or derive before you customize? Here are some hints to help you decide:

- A derived package is sufficient for minor changes to a package, such as switching out image files or adding a custom configuration.
- A derived package starts with the original package at it would install on an existing system, and then applies your custom changes.
- A shadow lets you recompile application source code with different options, or build the package with new requirements.
- A derived package version must match its parent package, while a shadow's version can increment over time.

After you have determined what kind of changes you need to make, use the following sections for instructions on deriving and shadowing.

5.1.1 Derive a Package

As previously stated, the process of creating a derived package starts by creating a full shadow of the original package, including the recipe and all other package source files. Then, you can check out the shadow, remove anything that should stay the same, modify the files that should be different, and adjust the recipe to add your customizations.

rBuild automates several of these tasks, and it is the recommended way to derive packages. You can also use `cvc shadow` and a series of manual steps. The sections that follow provide instructions for each of these options.

5.1.1.1 Derive an Encapsulated Package

Conary packages for encapsulated platforms like Red Hat Enterprise Linux are structured to obtain the RPM equivalent packages rather than to repackage the platform software in Conary. Originally, this meant that deriving packages using rbuild would not work for these platforms. Conary and rbuild have been updated so that it works much like traditional derivation, with the exception that files cannot be removed from the package.

When you attempt to derive you should discover what its parent source is and only attempt to derive the parent. You can discover the parent sources by running a command similar to the following example:

```
conary rq PACKAGENAME=centos.rpath.com@rpath:centos-5e  --info
```

Replace `PACKAGENAME` with the name of the package you want to find the parent source of.

Be sure to update to the latest version of the rbuild package, and then deriving a package from one of these encapsulated platforms should be similar to the `sudo` example shown here:

- `rbuild checkout --derive sudo`
- `cd sudo`
- Edit the recipe:

```
class SudoRecipe(DerivedCapsuleRecipe):
    name = 'sudo'
    version = '1.7.2p1_6.el5_5'

    def setup(r):
        # your modifications go here
```

- `cd ..`
- `rbuild build packages sudo --no-recurse`

You can also create new capsule packages. For more information, see *Encapsulate an Existing RPM*.

5.1.1.2 Derive with rBuild

If you're developing appliances with rBuilder, you should use rBuild when deriving packages. If you haven't yet, see how rBuild is used in the *rBuilder Package and Automation* guide associated with your version of rBuilder, published at *docs.rpath.com/rbuilder*. This document is an assumed prerequisite to the information in this section.

Use the `--derive` option with `rbuild checkout` to create the derived package. Change to the Development stage directory of your appliance checkout to run the command, and use the appropriate label or full version string to identify the parent branch from which you're deriving. The following shows deriving the splashy-theme package from the rPath Linux platform to the Example appliance:

```
$> cd example-1/Development
```

```
$> rbuild checkout --derive splashy-theme=conary.rpath.com@rpl:2
$> ls
group-example-appliance          splashy-theme
$> cd splashy-theme
$> ls
CONARY     _ROOT_     splashy-theme.recipe
```

rBuild automatically makes a checkout with your new derived package. Here's what you'll have in that checkout:

- *CONARY* state file, as in all packages and groups (Do not modify or remove this file.)
- *_ROOT_* directory with a copy of all the files as they would be installed on the target system (This represents the files that will be installed and managed by the final package build on the parent branch.)
- The package recipe file, recreated as a derived package recipe that uses the same package name and version

If you need to modify a file, copy it from the *_ROOT_* directory your package checkout alongside the recipe file. Then, make your modifications to that copy, and add an `r.addSource` line to your package recipe to add that file to the appropriate location on the target filesystem.

> **Leave _ROOT_ for reference**
>
> Do not make your changes directly in the *_ROOT_* directory. Leave that directory as the representation of the package which you are deriving.

After you make your adjustments, build your derived package as you would any other package for your appliance. Note that you may see a warning that you have not yet added the derived package to your appliance group recipe:

```
$> rbuild build packages splashy-theme
```

Be sure to add your derived package to your appliance. If you are editing your recipe directly (recommended), use `r.add` or `r.replace` as approriate. Even if the package was originally added automatically from your platform repository, you will need to use an `r.replace` line to indicate that the package should come from your appliance repository instead:

```
class OverrideRecipe(FactoryRecipeClass):
    def addRecipePackages(r):
        r.replace('splashy-theme')
```

See *Updating Package Versions for Derived Packages* for information about using newer versions of the parent branch over time, including examples of derived package recipes, and see *Merging Changes from Upstream* for instructions on bringing in upstream changes from the parent branch.

5.1.1.3 Derive with Conary (cvc)

If you're packaging for Conary outside of rBuilder's recommended appliance development work, you may also want to use Conary's cvc commands to derive the package. You can do this in three steps:

1. Use `cvc shadow` to shadow the package from the "parent" label to your own development label...
2. Check out the shadowed package in your Conary context where you are doing other packaging work.

3. Use `cvc remove` to remove any of the packaged files in the checkout that you do *not* need to modify from the original package.
4. Modify the files that you *do* need to modify, and add files if necessary.
5. Modify the recipe to change it to a derived package recipe, using the following steps:
    1. Remove any method calls not related to the modifications you made to the package files. Typically, this means you would remove all the method calls except for those that directly affect the files you modified.
    2. Add any method calls needed to apply the changes, such as *addSource()* lines to add new files.
    3. Change the inherited package recipe class to DerivedPackageRecipe. *DO NOT* modify the package name or version from the original recipe. This is critical to ensure that Conary can find the correct binary of the original package on the parent branch.

After you modify the checkout of your derived package, build and test it as you would any other package. See *Updating Package Versions for Derived Packages* for examples of derived package recipes, and see *Merging Changes from Upstream* for instructions on bringing in upstream changes from the parent branch.

To use your new derived package, be sure to install the package from your derived label rather from the parent label. Use *r.add()* or *r.replace()* as appropriate when adding the package to groups.

5.1.1.4 Updating Package Versions for Derived Packages

Because the derived package is a completely new package created from a shdow, there is no need to merge changes from the parent branch. However, when the parent branch releases a new version of the package, you may want to update the `version` line in your derived package recipe, and then rebuild the package to derive from that newer version:

```
class SplashyTheme(DerivedPackageRecipe):
    name = 'splashy-theme'
    version = '0.3.5'

    def setup(r):
        r.addSource('background.png', dest='%(datadir)s/splashy/themes/background.png')
        r.addSource('background.png', dest='/boot/extlinux/background.png')

class SplashyTheme(DerivedPackageRecipe):
    name = 'splashy-theme'
    version = '0.4.1'

    def setup(r):
        r.addSource('background.png', dest='%(datadir)s/splashy/themes/background.png')
        r.addSource('background.png', dest='/boot/extlinux/background.png')
```

See *Merging Changes from Upstream* for instructions on bringing in upstream changes from the parent branch.

5.1.2 Shadow a Package

When you use a full shadow instead of a derived package, your package build will perform the original build actions used to build the package, just with your modifications added. This is usually only necessary if you need to change the way the package builds, such as by adding compiler options.

rBuild is the recommended way to derive packages for those developing appliances with rBuilder. You can also use `cvc shadow`, especially if you are packaging separate from an appliance. The sections that follow provide instructions for each of these options.

5.1.2.1 Shadow with rBuild

If you're developing appliances with rBuilder, you should use rBuild when shadowing packages. If you haven't yet, see how rBuild is used in the *rBuilder Package and Automation* guide associated with your version of rBuilder, published at *docs.rpath.com/rbuilder*. This document is an assumed prerequisite to the information in this section.

Use the `--shadow` option with `rbuild checkout` to shadow the package to your appliance repository. Change to the Development stage directory of your appliance checkout to run the command, and use the appropriate label or full version string to identify the parent branch from which you're shadowing. The following shows shadowing the splashy package from the rPath Linux platform to the Example appliance:

```
$> cd example-1/Development
$> rbuild checkout -shadow splashy=conary.rpath.com@rpl:2
$> ls
group-example-appliance          splashy-theme        splashy
```

rBuild automatically makes a checkout with your new derived package. Change to the checkout to work with the package files, making the modifications you need.

After you make your adjustments, build your modified shadow just as you would any other package for your appliance. Note that you may see a warning that you have not yet added the shadowed package to your appliance group recipe:

```
$> rbuild build packages splashy
```

Be sure to add your modified shadow package to your appliance. If you are editing your recipe directly (recommended), use `r.add` or `r.replace` as approriate. Even if the package was originally added automatically from your platform repository, you will need to use an `r.replace` line to indicate that the package should come from your appliance repository instead:

```
class OverrideRecipe(FactoryRecipeClass):
    def addRecipePackages(r):
        r.replace('splashy-theme')
        r.replace('splashy')
```

See *Merging Changes from Upstream* for instructions on bringing in upstream changes from the parent branch.

5.1.2.2 Shadow with Conary (cvc)

If you're packaging for Conary outside of rBuilder's recommended appliance development work, you may also want to use Conary's cvc commands to shadow the package. You can do this in three steps:

1. Use `cvc shadow` to shadow the package from the "parent" label to your own development label...
2. Check out the shadowed package in your Conary context where you are doing other packaging work.
3. Modify the packaged files as needed, just as in other packaging work.

After you modify the checkout of your shadow, build and test it as you would any other package.

To use your new shadow, be sure to install the package from your shadowed label rather from the parent label. Use *r.add()* or *r.replace()* as appropriate when adding the package to groups.

See *Merging Changes from Upstream* for instructions on bringing in upstream changes from the parent branch.

5.2 Merging Changes from Upstream

As development continues on the parent branch from which you shadowed or derived, you may find a need to bring in those changes on your own development branch. Before merging, be sure to compare revisions of the package source between the target packages:

- **Compare Two Packages (conary) --** Use the `--file-versions` option with the `conary repquery` command to see what files are different between any two packages. In the following example, the user is querying *httpd:runtime* to see what is different from the parent branch and his own development branch, where he has a derived package. The output shows that the file *httpd.conf* has a derived version on *example.rpath.org@corp:devel* while the file *magic* is obtained from the parent branch (*conary.rpath.com@rpl:devel//1*):

  ```
  $> conary rq httpd:runtime=example.rpath.org@corp:devel --file-versions --full-
  versions
  /etc/httpd/conf/httpd.conf    /conary.rpath.com@rpl:devel//1//
  example.rpath.org@corp:devel/2.0.55-7.0.1-1
  /etc/httpd/conf/magic    /conary.rpath.com@rpl:devel//1//2.0.54-2-2
  ```

- **Compare Between Labels in the Repository (cvc) --** Use `cvc rdiff` to compare revisions of the same source between labels. The following example compares the latest committed versions of the shadow and parent:

  ```
  $> cvc rdiff splashy-theme example.rpath.org@corp:devel conary.rpath.com@rpl:2
  ```

- **Compare Between Your Checkout and a Label in the Repository (cvc) --** Use `cvc diff` to view the modifications made to a checked out copy of package source against those on a given label. Run the command from the local directory where you've checked out your package. The following example compares the current checkout directory with its project example:

  ```
  $> cvc diff conary.rpath.com@rpl:2
  ```

After you have compared the differences between your package and its parent, use `cvc merge` from your package checkout to merge revisions:

```
$> cvc merge
```

Add a revision argument if desired to target a specific package revision on the parent branch:

```
$> cvc merge 1.0.5
```

On rare occasions, merge conflicts can occur. When this happens, Conary creates a *.conflicts* file for each file that had conflicts. You can examine these files and the screen's output to troubleshoot the failed merge operation.

> ### Handling three-way merges
>
> Note that it is not possible to use `cvc diff` to preview a three-way merge before you do it. Instead, you can do a `cvc merge`, and then run `cvc diff` afterward to see if things merged as you needed.

If you're satisfied with the results, commit the changes with `cvc ci`. If you don't like the results, use `cvc revert` to revert your current checkout to its last committed state.

5.3 Cloning and Promoting

If you don't need to track and merge upstream changes from a parent branch, you can create a new package and copy the contents of the original package to start your development branch. You have two ways to do this:

- Use `rbuild checkout --new` followed by manual efforts to copy files to the new package.
- (recommended) Use `cvc clone` to create the cloned package, and then just check out the clone to start your modifications.

From that point on, the clone behaves as if you created it, so build and maintain it in the same way as your other packages. You will not be able to use the `cvc merge` operation on your clone.

Cloning creates a sibling relationship between the original package and the clone. In other words, if the original package is a shadow, the clone with share the package's parent branch. For a detailed explanation of this relationship, see the "Branching with Shadows and Clones" section in *Conary Versions and Flavors* available at *docs.rpath.com/conary*.

If you're using rBuild, you can promote your packages and groups through stages with the `rbuild promote` command. Conary also includes the command `cvc promote` for use outside of the rBuild structure. This cvc promote lets you to clone a group from one label to another, prompting Conary to also recurse the group and clone all the packages and components also included in that group.

The `fromLocation` and `toLocation` values used in the `cvc promote` command can be full branches, labels, branch names (begining with "@"), or namespaces (beginning with ":"). Some variations are:

```
$> cvc promote group-example example.rpath.org@corp:1-test--example.rpath.org@corp:1
$> cvc promote group-example @corp:1-test--@corp:1
$> cvc promote group-example :1-test--:1
```

Conary completes the missing parts of a label as follows:

- Using the `installLabelPath` value in the current Conary context for the `fromLocation`
- Using the `fromLocation` label for completing the `toLocation`

Use the following command to show the `installLabelPath` entry that applies in the scope of your current working directory:

```
$> conary config | grep installLabelPath
```

When you abbreviate the labels used in the command, Conary will return an error if the promoted item is not found on any of the labels in the `installLabelPath` that applies to the scope of your current working directory. In such cases, use the complete label to override the auto-completion by Conary.

Limitations for cloning and promoting

Promote and clone have a limitation with respect to the parent-child and sibling relationships revealed in a the full version string for the component, package, or group you're cloning. Developers can clone between siblings, such as:

- */branchA* to */branchB*
- */parentZ//branchA* to */parentZ//branchB*

They can also clone upstream to parents, such as:

- */parentZ//branchB* to */parentZ*

...and upstream to siblings of parents, such as:

- */parentZ//parentY//branchC* to */parentZ//parentX*
- */parentZ//parentY//branchC* to */parentZ*

However, developers cannot clone between shadow branches that do not share a parent, such as */parentZ//branchA* to */parentW//branchB*).

Appendix A. Recipe Actions, Macros, and Variables

When you're creating packages and groups for Conary package management, such as for appliances in rBuilder, you can use the *Conary Application Programming Interface (API)* as a reference for the actions you can include as part of each recipe. All recipes start by inheriting from these basic recipe actions, even if you load and use another recipe or superclass.

How can you navigate the Conary API to find all these actions?

No worries. This document outlines the recipe actions you can use, along with links to the *Conary API* for a complete listing of arguments and examples. Use the following sections as a reference, and keep in mind the following important points:

- In package recipes, no matter what order your actions appear, they will always occur in this order when you build the package:

    1. Source actions
    2. Build actions
    3. Policy actions

    The sections that follow separate out the actions that fall into these three types.

- You can use any of these actions in any of your package recipes. However, if you're loading another recipe or superclass, such as those throughout the *Recipe Types and Templates* at *docs.rpath.com/conary*, know what actions the superclass has already done for you. Any of the public methods in that superclass are also actions you can use in your recipe, supplementing the Conary actions detailed here.

> **Quick reference with `cvc explain`**
>
> If you need a quick glance at the API documentation for a particular recipe action, you don't have to open the Web pages to find what you want. Just use the `cvc explain` command followed by the recipe action, such as in the following example:
>
> ```
> $> cvc explain addSource
> ```
>
> Scroll through the documentation as needed to find the information you need, and then press "q" to quit.

A.1 Package and Group Recipe Classes

Conary packages all inherit from base class `PackageRecipe`, though Conary provides all of the following recipe classes:

Table A.1. Package Recipe Classes

Class Name	Description and API Link
PackageRecipe	Base recipe class with all the essential requirements for Conary recipes
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.packagerecipe.PackageRecipe-class.html*
BuildPackageRecipe	Uses additional build requirements such as `grep` and `sed`, beyond those described in *Build Requirements and Search Path*
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.packagerecipe.BuildPackageRecipe-class.html*
CPackageRecipe	Used for most recipes built from C source code; inherits from `BuildPackageRecipe`; to discover the additional actions you can can use to simplify your code when using this recipe class, see "C Source Code" in *Recipe Types and Templates* at *docs.rpath.com/conary*
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.packagerecipe.CPackageRecipe-class.html*
AutoPackageRecipe	Used for recipes built from C source code which can use the `auto*` tools such as `automake` and `autoconf`; inherits from `CPackageRecipe`; to discover the additional actions you can can use to simplify your code when using this recipe class, see "C Source Code" in *Recipe Types and Templates* at *docs.rpath.com/conary*
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.packagerecipe.AutoPackageRecipe-class.html*

Siblings to PackageRecipe in the Conary source code include the following classes. See their descriptions for links to further information:

Table A.2. Other Recipe Classes

Class Name	Description and API Link
GroupRecipe	Class with variables and actions for creating Conary groups, used to install and manage a selection components, packages, and other groups; for the variables and actions specific to your group recipes, see *Group Variables and Actions*
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.grouprecipe.GroupRecipe-class.html*
DerivedPackageRecipe	Class used to reference the binary package build of another package, and then add any recipe actions to perform on the files managed by that package; for details on how to use this class, see the "Shadowing and Deriving" section of *Copying and Customizing Existing Conary Packages* at *docs.rpath.com/conary*

Class Name	Description and API Link
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.derivedrecipe.DerivedPackageRecipe-class.html*
RedirectRecipe	Class used to point one package or group to one or more other packages or groups; for details on how to use this class, see the "Redirect Packages and Groups" section of *Recipe Types and Templates* at *docs.rpath.com/conary*
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.redirectrecipe.RedirectRecipe-class.html*
FilesetRecipe	Not currently in active development or use; see the API doc for more information.
	API Doc: *http://cvs.rpath.com/conary-docs/conary.build.filesetrecipe.FilesetRecipe-class.html*

A.2 Package Variables and Actions

The `PackageRecipe` class includes all the details outlined in the following sections. Be sure to note any additional variables and actions added by any subclasses or superclasses you choose to use in your recipe code. For Conary groups, see *Group Variables and Actions* instead.

Each package must provide a value for the name and version of your package:

```
    name = 'example'
    version = '1.0a.7'
```

Beyond these variables, you can also have build requirements (`buildRequires`) as described in the next section.

A.3 Build Requirements and Search Path

Build requirements are components that are needed to perform certain build operations on your package, but that are not added to the package. For example, C source code may require *gcc:runtime* at build time to compile the source code, though the final application (and package) may not need *gcc:runtime* to run. Conary has a `baseRequiresRecipe` set of automatic build requirements for every recipe so that you don't have to rewrite them in your own recipe:

When you build a package, Conary might display a list of any additional build requirements it needs for its build process. Add these to the `buildRequires` variable in your recipe to include the appropriate components. This value will just add to the default list from Conary. In rare cases, you may need to add some build requirements not reported by Conary:

```
class Example(PackageRecipe):
    name = 'example'
    version = '1.0'

    buildRequires = [ 'perl:runtime', 'another:component' ]
```

You can also clear some build requirements you know you don't need by using `clearBuildRequires`:

```
class Example(PackageRecipe):
    name = 'example'
    version = '1.0'
    clearBuildRequires('tar:runtime', 'sed:runtime')
```

If no autoLoadRecipes options are specified, then the recipe classes used are the ones built into conary (in conary/build/*recipe.py modules). However, you can specify *autoLoadRecipes* configuration lines in your conaryrc to override the built-in classes. When using rBuild / *product-definition*, you get this data from your platform. For example, CentOS has this in the platform definition:

```
            <autoLoadRecipes>
                <autoLoadRecipe troveName="group-superclasses"
 label="centos.rpath.com@rpath:centos-5"/>
            </autoLoadRecipes>
```

that group contains:

```
            autopackage=1-1-1
            baserequires=1-2-1
            buildpackage=1-6-1
            cpackage=1-6-1
            derived=0-2-1
            fileset=0-2-1
            group=0-3-1
            groupinfo=1-1-1
            package=1-8-1
            redirect=0-1-1
            userinfo=1-1-1
```

To investigate the behavior of a given recipe class in CentOS, you could check it out as the following example shows:

```
            $ cvc co package=centos.rpath.com@rpath:centos-5 --dir /tmp/centos-package-
recipe
            $ cat /tmp/centos-package-recipe/package.recipe
            #
            # Copyright (c) 2008 rPath, Inc.
            # This file is distributed under the terms of the MIT License.
            # A copy is available at http://www.rpath.com/permanent/mit-license.html
            #

            from conary.build.packagerecipe import SourcePackageRecipe

            class PackageRecipe(SourcePackageRecipe, BaseRequiresRecipe):
            name = 'package'
            version = '1'
            abstractBaseClass = 1
            buildRequires = [
            'bzip2:runtime',
            'gzip:runtime',
            'tar:runtime',
```

```
'cpio:runtime',
'patch:runtime',

# Add for perl dep discovery
'perl:lib',

# Add for initscript tagging
'chkconfig:runtime',
]
```

Note this recipe inherits from baseRequires.

By default, rBuild, rMake, and Conary will use your current scope of Conary configuration to determine where to find each build requirement. rBuild sets this configuration for you automatically while rMake and Conary use information in the manually-maintained *.conaryrc* file in your home directory. There are two approaches to adding build requirements that are not in your search path:

- In your `buildRequires`, use the label on which a component resides, minus the repository hostname for the label, when you specify the component (everything from the "@" to the end of the label):

  ```
  buildRequires = [ 'perl:runtime', 'another:component=@corp:resource-1' ]
  ```

- Add to the `installLabelPath` in the Conary configuration of your build environment, especially in the specific context you have set up if you are using rMake and Conary without rBuild (see *Conary:Contexts* at *wiki.rpath.com*).

The path used to find your build requirements is related to your build environment for packages and groups, but your *search path* is where the appliance ultimately finds the finished packages for your appliance. Control the search path for your appliance in the appliance contents editor in rBuilder. See *Append Your Search Path* in the *rPath Packaging Guide* for more information.

There are two ways to cause a policy not to impact the files in a package:

- Use an exception that matches everything. The following example would effectively disable the calculation of PHP requirements for all files in a package:

  ```
  r.PHPRequires(exceptions='.*')
  ```

- Delete the policy entirely from your recipe class. The following example would delete the PHPRequires policy:

  ```
  del r.PHPRequires
  ```

In some cases, the second option gives better performance because the first option has to iterate through the file list, make function calls and perform the regular expression match for each file. However, you should examine a policy carefully before deleting it, as deleting a policy that another policy is dependent on can break the dependency chain.

A.4 Macros

To provide the greatest modularity for code, and avoid hard-coding directory paths and other strings that could change if you change your platform or other build conditions. Instead, use a *macro* that will be replaced with the appropriate value at build time. Conary has several built-in macros in */etc/conary/macros*, outlined in the sections that follow:

- *Directory Path Macros* expand to directory paths expected on your build system, and on the target/destination system where your package or group is ultimately installed. These are used more often than other macros.

- *Executable and Option Macros* expand to the commands and/or options common on most Linux systems, typically used to direct a package build on your build system.
- *Build and Cross-compile Macros* expand to useful strings used when cross-compiling, meaning that the recipe is packaging from source and needs to compile that source for a target platform does not resemble your build system (such as embedded systems).

You can also reate and use your own macro. The following code creates and uses a macro called `exampledir` to represent the directory where software files are unpacked:

```
r.macros.exampledir = '%(servicedir)s/%(name)s'
r.addArchive('%(name)s-%(version)s.tgz', dir='%(exampledir)s')
r.SetModes('%(exampledir)s/Settings.php', 0644)
```

A.4.1 Directory Path Macros

Directory path macros expand to directory paths expected on your build system, and on the target/destination system where your package or group is ultimately installed. Use the following directory path macros in your recipes where appropriate:

Table A.3. Directory Path Macros

Macro	Typically expands to...	Usage Tips and Examples
%(bindir)s	`%(exec_prefix)s/bin` (*/usr/bin* in most Linux systems)	This is the directory of executables that are available to users on the underlying system.
		`r.Install('example.exe','%(bindir)s/')`
%(builddir)s	The `buildPath` value in Conary configuration in the context (current working directory) where you're running your package build command; to check this value, use: `conary config | grep buildPath`	This is the directory defined for Conary's source building/compiling operations, primarily when packaging from source code. Source code is unpacked to `%(builddir)s`, and you can use the macro where appropriate to reference your unpacked source.
		`r.Environment('EXAMPLEPATH', '%(builddir)s/example')`
%(buildlogpath)s	`%(debugsrcdir)s/buildlogs/%(name)s-%(version)s-log.bz2` (`/usr/src/debug/buildlogs/<name>-<version>-log.bz2`)	This location is used internally by Conary. This macro should not be changed in your recipes, and it is not typically explicitly specified.

Macro	Typically expands to...	Usage Tips and Examples
%(cachedir)s	`%(localstatedir)s/cache` (*/var/cache* in most Linux systems)	This is the location for cached application data.
		`r.MakeDirs('%(cachedir)s/{local,X11R6}')`
%(datadir)s	`%(prefix)s/share` (*/usr/share* in most Linux systems)	The location where architecture-independent data resides. This directory contains subdirectories and data sharable among all architectures.
		`r.ComponentSpec('apidox', '%(datadir)s/%(qt)s/doc/')`
%(debuglibdir)s	`/usr/lib/debug`	This location is used in assembling packages for a `:debuginfo` component. This macro should not be changed in your recipes, and it is not typically explicitly specified.
%(debugsrcdir)s	`/usr/src/debug`	This location is used in assembling packages for a `:debuginfo` component. This macro should not be changed in your recipes, and it is not typically explicitly specified.
%(destdir)s	Set as needed by Conary based on the value of `%(builddir)s`	This is the installation directory used to install package files during a package build, imitating the files' organization on the target filesystem.
		`r.MakeInstall('PREFIX=%(destdir)s/%(prefix)s')`
%(docdir)s	`%(datadir)s/doc` (*/usr/share/doc* in most Linux systems)	This location contains miscellaneous documentation in various formats used by applications on the system.
		`r.ComponentSpec('doc', '%(docdir)s/.*')`
%(essentialbindir)s	*/bin* on a system that follows the Linux filesystem hierarchy	This is the directory for essential system binaries. This directory contains executables used for critical system operations, such as booting and mounting partitions.
		`r.Symlink('bash', '%(essentialbindir)s/sh')`
%(essentialsbindir)s	*/sbin* on a system that follows the Linux filesystem hierarchy	This is the location of essential system administration binaries. This directory contains executables used by the root user and utilites that run as the root user.
		`r.Move('/dev/MAKEDEV', '%(essentialsbindir)s/')`

Macro	Typically expands to...	Usage Tips and Examples
%(essentiallibdir)s	`/%(lib)s` (*/lib* or */lib64* in most Linux systems, depending on the architecture for which the recipe is built)	This is the directory for essential shared data files. This directory subdirectories, data files, and shared libraries.
		`r.MakeDirs('%(essentiallibdir)s/firmware')`
%(exec_prefix)s	`%(prefix)s` (*/usr* in most Linux systems)	This is the installation prefix for architecture-dependent files. This directory contains subdirectories of architecture-dependent executables, libraries, and other files.
		`r.ManualConfigure('--prefix=%(prefix)s --exec-prefix=%(exec_prefix)s')`
%(groupinfodir)s	`%(sysconfdir)s/conary/groupinfo` (*/etc/conary/groupinfo* on most Linux systems)	This location has Conary user group info used to provide Linux system group information to `info` recipes (see *Creating System Users and Groups with Info Packages* at *docs.rpath.com/conary*).
		`r.Install('samplegroup', '%(groupinfodir)s/')`
%(includedir)s	`%(prefix)s/include` (*/usr/include* in most Linux systems)	This is the location that contains header files used by applications on the system.
		`r.Symlink('%(x11includedir)s/X11', '%(includedir)s/X11')`
%(infodir)s	`%(datadir)s/info` (*/usr/share/info* in most Linux systems)	This is the primary location for the GNU `info` system. This directory contains subdirectories and compressed GNU info pages.
		`r.InitialContents('%(infodir)s/dir')`
%(initdir)s	`%(sysconfdir)s/init.d` (*/etc/init.d* in most Linux systems)	This indicates the location for `init` scripts used by `SysVinit`.
		`r.TagSpec('initscript', '%(initdir)s/')`
%(krbprefix)s	`%(exec_prefix)s/kerberos` (*/usr/kerberos* in most Linux systems)	This is the installation location for `kerberos`.
		`r.Remove('%(krbprefix)s/man/cat*', recursive=True)`
%(lib)s	*lib* or *lib64*	This is just a text replacement used in other directory paths that is determined by how the recipe is built, specifically whether or not it is built with the `x86_64` architecture flavor specification.

Macro	Typically expands to...	Usage Tips and Examples
%(libdir)s	`%(exec_prefix)/%(lib)s` (*/usr/lib* or */usr/lib64* in most Linux systems, depending on the architecture for which the recipe is built)	This is the directory for shared data files. This directory subdirectories, data files, and shared libraries.
		`r.Remove('%(libdir)s/libss.a')`
%(libexecdir)s	`%(exec_prefix)s/libexec` (*/usr/libexec* in most Linux systems)	This is the location of helper binaries. This directory contains small helper executables called by other applications.
		`r.SetModes('%(libexecdir)s/pt_chown', 0755)`
%(prefix)s	*/usr* on a system that follows the Linux filesystem hierarchy	This is the system resources directory, which contains subdirectories of executables, libraries, documentation, and other files which do not change during normal operations.
		`r.Make("PREFIX=%(prefix)s")`
%(localstatedir)s	*/var* on a system that follows the Linux filesystem hierarchy	This is the dynamic files location. This directory contains subdirectories of files that change with the system state, such as temporary files and log files.
		`r.Make Dirs('%(localstatedir)s/run/radvd/')`
%(mandir)s	`%(datadir)s/man` (*/usr/share/man* in most Linux systems)	This is the location of the online manual files used by the `man` utility on Linux systems. This directory contains subdirectories associated with each manual section, and each manual file is installed in its appropriate section.
		`r.Install('anacron.8', '%(mandir)s/man8/')`
%(sbindir)s	`%(exec_prefix)s/sbin` (*/usr/sbin* in most Linux systems)	This is the location of system administration binaries. This directory contains executables used by the root user for system administration tasks such as network configuration.
		`r.Install('inputattach', '%(sbindir)s/')`
%(servicedir)s	*/srv* on a system that follows the Linux filesystem hierarchy	This is the location for data related to services offered by the system. This directory contains subdirectories, each with data associated with a given service.
		`r.SetModes('%(servicedir)s/example', 0640)`

Macro	Typically expands to...	Usage Tips and Examples
%(sharedstatedir)s	`%(prefix)/com` (*/usr/com* on most Linux systems)	The location for architecture-independent data files which are modified by applications when they run.
		`r.Make('--sharedstatedir=%(sharedstatedir)s')`
%(sysconfdir)s	*/etc* on a system that follows the Linux filesystem hierarchy	This is the system configuration directory, which contains subdirectories and system-wide configuration files.
		`r.Create('%(sysconfdir)s/example/example.conf')`
%(tagdatadir)s	`%(datadir)s/conary/tags` (*/usr/share/conary/tags* in most Linux systems)	This is the location of the data used by Conary tag handlers. This macro should not be modified in your recipes.
		`r.addSource(catalog, dest='%(tagdatadir)s/xml-catalog/desc.d/')`
%(tagdescriptiondir)s	`%(sysconfdir)s/conary/tags` (*/etc/conary/tags* on most Linux systems)	This is the location of the Conary tag description files. This macro should not be modified in your recipes.
		`r.addSource('info-file.tagdescription', macros=True, dest='%(tagdescriptiondir)s/info-file'))`
%(taghandlerdir)s	`%(libexecdir)s/conary/tags` (*/usr/libexec/conary/tags* in most Linux systems)	This is the location of the Conary tag handler files. This macro should not be modified in your recipes.
		`r.Requires('grep:runtime', '%(taghandlerdir)s/')`
%(testdir)s	`%(localstatedir)s/conary/tests` (*/var/conary/tests* on most Linux systems)	This macro is specified internally by the `r.TestSuite()` recipe action
%(thisdocdir)s	`%(docdir)/%(name)s-%(version)s` (*/usr/share/doc/<package_name>-<package_version>* on most Linux systems)	This is a location in the documentation directory that contains documentation specific to your package, using the `name` and `version` variables from your package recipe in the subdirectory name.
		`r.Move('%(docdir)s/html/*', '%(thisdocdir)s/html/')`
%(thistestdir)s	`%(testdir)/%(name)s-%(version)s` (*/var/conary/tests/<name>-*	This is a location in the test suite directory that contains test data specific to your package, using the `name` and

Macro	Typically expands to...	Usage Tips and Examples
	<*version*> on most Linux systems)	`version` variables from your package recipe in the sub-directory name.
		`r.Replace(('^#!example', '#! %(thistestdir)s/example')`
%(userinfodir)s	`%(sysconfdir)s/conary/userinfo` (*/etc/conary/userinfo* on most Linux systems)	This location has Conary user info used to provide user information to `info` recipes (see *Creating System Users and Groups with Info Packages* at *docs.rpath.com/conary*).
		`r.Install('jqsample', '%(userinfodir)s/')`
%(x11prefix)s	`%(exec_prefix)s/X11R6` (*/usr/X11R6* in most Linux systems)	This is the installation location for `X11`.
		`r.MakeDirs('%(x11prefix)s/lib/X11/app-defaults')`

A.4.2 Executable and Option Macros

Executable and option macros expand to the commands and/or options you might use. The default values for these macros are commands and options common on most Linux systems, and which you might use to direct a package build on your build system. Use the following executable and option macros in your recipes where appropriate:

Table A.4. Executable and Option Macros

Macro	Default Value	What This Represents
%(cc)s	`gcc`	C compiler
%(cxx)s	`g++`	C++ compiler
%(cxxflags)s	(no default value)	Flags specific to %(cxx)s
%(cflags)s	`%(optflags)s %(dbgflags)s`	Combined C compiler optimization and debug flags
%(cppflags)s	(no default value)	C-processor options
%(dbgflags)s	`-g`	Assembly of %(cflags)s and %(cppflags)s macros
%(debugedit)s	`debugedit`	Utility for generating *debuginfo* files
%(optflags)s	`-O2`	C compiler optimization options
%(os)s	linux	Operating system
%(ldflags)s	`%(dbgflags)s`	Linker flags

Macro	Default Value	What This Represents
%(mflags)s	(no default value)	`make` flags
%(monodis)s	`%(bindir)s/monodis`	Utility for determining Mono dependencies
%(parallelmflags)s	(no default value)	parallel `make` flags
%(strip)s	`eu-strip`	Method for stripping symbols from object files (macro not typically used explicitly)
%(strip_archive)s	`strip -g`	Method for stripping debugging symbols from object files (macro not typically used explicitly)
%(sysroot)s	(no default value)	Alternate system root for libraries and executables when cross-compiling

A.4.3 Build and Cross-compile Macros

Build-time macros expand to complete certain Conary configuration information. You should never reset these macros in your recipe. *Cross-compile macros* expand to useful strings used when cross-compiling, meaning that the recipe is packaging from source and needs to compile that source for a target platform does not resemble your build system (such as embedded systems).

Use the following build-time and cross-compile macros in your recipe where appropriate:

Table A.5. Build-time Macros

Macro	Default Value	Usage Tips and Examples
%(major_version)s	same value as `%(version)s`	Isolates the major version fo software from the Conary-provided `version` value based on expected structure of that version. In Python, the operation performed on the version is `'.'.join(r.version.split('.')[0,2])`. This prevents the copy/paste issues from recipe to recipe and allows easier maintenance for those unfamiliar with Python.
%(buildbranch)s	(no default value)	Repository branch associated with the package or group you're building
		`/example.rpath.org@corp:example-1-devel//example-1/`
%(buildlabel)s	(no default value)	Label of the branch associated with the package or group you're building
		`example.rpath.org@corp:example-1`

Table A.6. Cross-compile Macros

Macro	Default Value	Usage Tips and Examples
%(buildcc)s	`%(cc)s`	This is the C compiler that will create executables that you can use on the build system's architecture
		`r.Run('%(buildcc)s %(optflags)s -o foo foo.c')`
%(buildcxx)s	`%(cxx)s`	This is the C++ compiler that will create executables that you can use on the build system's architecture
		`r.Configure(buildconf, bootstrapFlags=, preConfigure='CC=%(buildcc)s CXX=%(buildcxx)s')`
%(build)s	`%(buildarch)s-%(buildvendor)s-%(buildos)s`	This is the combined macro that includes values related to the build system. This is used to define values in the `%(buildcc)s` macro.
%(buildos)s	`ulinux`	This is the build system's operating system. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(buildvendor)s	`unknown`	This is the build system's vendor. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(host)s	`%(hostarch)s-%(hostvendor)s-%(hostos)s`	This is the combined macro that includes values related to the host system. This is used to define values in the `%(buildcc)s` macro.
%(hostos)s	`linux`	This the operating system of the host system, where the compiled application will run. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(hostvendor)s	`unknown`	This the vendor of the host system, where the compiled application will run. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(target)s	`%(targetarch)s-%(targetvendor)s-%(targetos)s`	This is the combined macro that includes values related to the target system. This is used to define values in the `%(buildcc)s` macro.
%(targetos)s	`linux`	This the operating system of the target system, where your completed package or group will be installed and

Macro	Default Value	Usage Tips and Examples
		managed. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(targetvendor)s	`unknown`	This the vendor of the target system, where your completed package or group will be installed and managed. This is not specified explicitly, but rather used to define values in the `%(buildcc)s` macro.
%(crossdir)s	`cross-target-%(target)s`	This is the cross-compiling tools installation directory.
%(crossprefix)s	`/opt/%(crossdir)s`	The cross-compiler location with directory prefix.
		`%(crossprefix)s` might expand to something like */opt/cross-target-x86_64*
%(headerpath)s	`%(sysroot)s/%(includedir)s`	This is the location of the cross-compile target header files.
		`%(headerpaths)s` might expand to something like */opt/cross-target-x86_64/sys-root/usr/include*
%(sysroot)s	`%(crossprefix)s/sysroot`	This is the location of cross-compiled libraries and executables.
		`%(sysroot)s` might expand to something like */opt/cross-target-x86_64/sys-root*

A.5 Source Actions

Use a combination of the following actions to obtain and prepare the files to use when building your package. Use macros *Macros* where applicable to prevent hard-coding certain information such as directory paths. (To complete the URL for the links, start with: **http://cvs.rpath.com/conary-docs/**)

Table A.7. Source Actions for Package Recipes

Action	Description and API Link
r.addAction	Perform the shell command provided in the first pass of the build; similar to `r.Run()`, but performed during the prep stage of the package build
	conary.build.source.addAction-class.html
r.addArchive	Add an archive, such as a tarball or zip file, unpacking it into the appropriate directory
	conary.build.source.addArchive-class.html

Action	Description and API Link
r.addPatch	Apply a patch
	conary.build.source.addPatch-class.html
r.addSource	Copy a file into the build directory or a specific destination directory
	conary.build.source.addSource-class.html
r.addBzrSnapshot	Checks out contents from a Bazaar version control system
	conary.build.source.addBzrSnapshot-class.html
r.addCvsSnapshot	Checks out contents from a CVS version control system
	conary.build.source.addCvsSnapshot-class.html
r.addGitSnapshot	Checks out contents from a Git version control system
	conary.build.source.addGitSnapshot-class.html
r.addMercurialSnapshot	Checks out contents from a Mercurial version control system
	conary.build.source.addMercurialSnapshot-class.html
r.addSvnSnapshot	Checks out contents from a Subversion version control system
	conary.build.source.addSvnSnapshot-class.html
r.addPostInstallScript	Specifies the post install script for a trove
	conary.build.source.addPostInstallScript-class.html
r.addPostRollbackScript	Specifies the post rollback script for a trove
	conary.build.source.addPostRollbackScript-class.html
r.addPostUpdateScript	Specifies the post update script for a trove
	conary.build.source.addPostUpdateScript-class.html
r.addPreRollbackScript	Specifies the pre rollback script for a trove
	conary.build.source.addPreRollbackScript-class.html
r.addPreUpdateScript	Specifies the pre update script for a trove
	conary.build.source.addPreUpdateScript-class.html

A.6 Build Actions

Use a combination of these actions to direct the installation of the software and the building of the package. Use macros *Macros* where applicable to prevent hard-coding certain information such as directory paths. (To complete the URL for the links, start with:)

> **Note**
>
> These are generally the actions in the *BuildCommand* class and its subclasses.

Table A.8. Build Actions for Package Recipes

Action	Description and API Link
r.Ant	Execute the `ant` utility
	conary.build.build.Ant-class.html
r.Automake	Runs the `aclocal`, `autoconf`, and `automake` commands
	conary.build.build.Automake-class.html
r.ClassPath	Set the `CLASSPATH` environment variable
	conary.build.build.ClassPath-class.html
r.CompilePython	Generate compiled Python files (*.pyc* and *.pyo* files)
	conary.build.build.CompilePython-class.html
r.Configure	Run an `autoconf` configure script; provides many common arguments set to values provided by system macros; if the common arguments don't work for your code, use `r.ManualConfigure()` instead and add your replacement values.
	conary.build.build.Configure-class.html
r.ConsoleHelper	Set up `consolehelper` symbolic links, control files, and dependency for an application
	conary.build.build.ConsoleHelper-class.html
r.Copy	Copy files from one directory to another without changing their mode
	conary.build.build.Copy-class.html
r.Cmake	Run a `cmake` configure script
	conary.build.build.CMake-class.html
r.Create	Create a file, either empty or with some initial contents

Action	Description and API Link
	conary.build.build.Create-class.html
r.Desktopfile	Install desktop (`.desktop`) files in their appropriate location
	conary.build.build.Desktopfile-class.html
r.Doc	Install documentation files to ensure Conary recognizes and treats the files as documentation; if you use `r.PackageSpec()` or `r.ComponentSpec()` actions on the same files, they will override any `package=` arguments in `r.Doc()` actions
	conary.build.build.Doc-class.html
r.Environment	Set an environment variable; follows the automatic setting of Conary macros as described in *Macros*
	conary.build.build.Environment-class.html
r.Glob	Translate a given glob statement into a regular expression, substituting macros
	conary.build.action.Glob-class.html
r.Group	*ONLY for group info- packages*; see *Creating System Users and Groups with Info Packages* at *docs.rpath.com/conary* for more information
	http://cvs.rpath.com/conary-policy/stubs.Group-class.html
r.IncludeLicense	Include a GPL or CPL license, taking either a directory structure of licenses or a single license file, normalizing its contents, and placing it in */usr/share/known-licenses*, to which Conary applies certain policy actions
	conary.build.build.IncludeLicense-class.html
r.Install	Copy files from one location to another and sets their mode (permissions)
	conary.build.build.Install-class.html
r.JavaCompile	Run the Java compiler on a specified file or directory, providing a different compiler command if you need something other than `javac`, and providing arguments to that command if necessary
	conary.build.build.JavaCompile-class.html
r.Ldconfig	Run the `ldconfig` utility in a specified directory; used mainly when the software does not set up all appropriate symbolic links for a shared library
	conary.build.build.Ldconfig-class.html

Action	Description and API Link
r.Link	Create a hard link so that two directory paths affect a single directory or file; rarely used, and should be avoided in favor of symbolic links when possible (see the `r.Symlink()` action); note that with hard links, the original path and the link path identify the same item, and actions on either path affect the same data, including corruption or deletion
	conary.build.build.Link-class.html
r.Make	Run the `make` utility with system defaults, or with any forced flag overrides and other arguments you pass to the recipe action
	conary.build.build.Make-class.html
r.MakeDirs	Create directories, and set the mode (permissions) on those directories
	conary.build.build.MakeDirs-class.html
r.MakeFIFO	Create a named pipe for file-in/file-out (FIFO) operations; for more information on pipes and FIFOs, see *resources at Gnu.org*
	conary.build.build.MakeFIFO-class.html
r.MakeInstall	Run `make install` and set the install target (destination) directory
	conary.build.build.MakeInstall-class.html
r.MakeParallelSubdir	When you run a parallel make, run `make` with the system default for `parallelmflags` applied only to sub-make processes
	conary.build.build.MakeParallelSubdir-class.html
r.MakePathsInstall	Run make when there is no single functional destination directory, but the Makefile honors standard variables to make a destination directory install successful; set the destination directory as an argument in this action
	conary.build.build.MakePathsInstall-class.html
r.ManualConfigure	Use this instead of `r.Configure()` if you need to explicitly provide all the arguments to the command, not just supplemental arguments
	conary.build.build.ManualConfigure-class.html
r.Move	Move files from one directory to another; the mode (permissions) is preserved
	conary.build.build.Move-class.html
r.PythonSetup	Run `setup.py` using the *python-setuptools* library to install without building an *.egg* file

Action	Description and API Link
	conary.build.build.PythonSetup-class.html
r.Regexp	Translate a regular expression to use Conary macros
	conary.build.action.Regexp-class.html
r.Remove	Remove files and directories, including recursing any given directory; can take globs, but not regular expressions
	conary.build.build.Remove-class.html
r.Replace	Substitute text in a file; ideal to make minor adjustments to configuration or code that are specific to your packaging needs
	conary.build.build.Replace-class.html
r.Run	Run a specified shell command during the build process for your package
	conary.build.build.Run-class.html
r.SetModes	Set the mode (permissions) on files or directories; provide multiple files and directories together if you need to set them all to the same mode
	conary.build.build.SetModes-class.html
r.SGMLCatalogEntry	Add an entry to the SGML catalog file
	conary.build.build.SGMLCatalogEntry-class.html
r.SupplementalGroup	*ONLY for group info- packages*; see *Creating System Users and Groups with Info Packages* at *docs.rpath.com/conary* for more information
	http://cvs.rpath.com/conary-policy/stubs.Group-class.html
r.Symlink	Create a symbolic link between files or directories
	conary.build.build.Symlink-class.html
r.TestSuite	Create a script to that can run the test suite for the packaged software as if it is packaged as part of the software; not actively used by Conary developers
	conary.build.build.TestSuite-class.html
r.User	*ONLY for user info- packages*; see *Creating System Users and Groups with Info Packages* at *docs.rpath.com/conary* for more information
	http://cvs.rpath.com/conary-policy/stubs.User-class.html

Action	Description and API Link
r.XIntedService	Create a file in */etc/xinted.d/* for running an application with the `xinted` daemon
	conary.build.build.XInetdService-class.html
r.XMLCatalogEntry	Add an entry to the XML catalog file
	conary.build.build.XMLCatalogEntry-class.html

A.7 Policy Actions

Use a combination of these actions override Conary's default policy about how the package installs and behaves. Use macros *Macros* where applicable to prevent hard-coding certain information such as directory paths.

The first of two tables includes built-in policy actions in the *Policy* class and its subclasses. (To complete the URL for the links, start with: **http://cvs.rpath.com/conary-policy/**)

Table A.9. Built-in Policy Actions for Package Recipes

Action	Description and API Link
r.ByDefault	Override Conary's default determination for what components should be installed by default when the package is installed; specify inclusions or exceptions
	conary.build.packagepolicy.ByDefault-class.html
r.ComponentProvides	Sets the package to explicitly provide its name; use this to provide optional capability flags that can be used with that name
	conary.build.packagepolicy.ComponentProvides-class.html
r.ComponentRequires	Create dependencies between components from different packages, so that one of the components from your package will not install without also installing one or more specific components from another package; if necessary, override Conary's default component assignments with the `r.ComponentSpec()` policy action
	conary.build.packagepolicy.ComponentRequires-class.html
r.ComponentSpec	Override Conary's default component assignment by specifying the components that should manage each of the files installed by your package; use a `catchall` argument for this action to catch any unspecified files in a specific component
	conary.build.packagepolicy.ComponentSpec-class.html

Action	Description and API Link
r.Config	Mark one or more files as a configuration file so that Conary will manage the file like other configuration files (with merges instead of overwrites during updates)
	conary.build.packagepolicy.Config-class.html
r.ExcludeDirectories	Override Conary's default behavior of deleting empty directories by specifying those directories as arguments to this action; this is not necessary if your recipe already uses `r.SetModes()` or `r.Ownership()` on the directory
	conary.build.packagepolicy.ExcludeDirectories-class.html
r.Flavor	Mark files as associated with a given flavor specification, or prevent a file from being marked as flavor-specific
	conary.build.packagepolicy.Flavor-class.html
r.InitialContents	Override Conary's default marking of initial contents files with an explicit list of initial contents, specificed as arguments in this action; initial contents files are installed once, during a package install or update, if they did not already exist on the system, but those files will never be updated to newer versions if they still exist when the package is updated
	conary.build.packagepolicy.InitialContents-class.html
r.LinkCount	Control what directories can or cannot hard-link between other directories
	conary.build.packagepolicy.LinkCount-class.html
r.LinkType	Indicate that only regular, non-config files can have hard links
	conary.build.packagepolicy.LinkType-class.html
r.MakeDevices	Create device nodes on the system for use by the package; pass the node details in as arguments to this action
	conary.build.packagepolicy.MakeDevices-class.html
r.Ownership	Set the user and group ownership for a file or directory, overriding Conary's default of `root:root`
	conary.build.packagepolicy.Ownership-class.html
r.PackageSpec	If your package recipe creates two or more packages, specify what package (and, optionally, component) should install and manage specific files; if the package recipe creates just one package, use the `r.ComponentSpec()` action instead

Action	Description and API Link
	conary.build.packagepolicy.PackageSpec-class.html
r.Provides	Mark files as providing certain features or characteristics, or prevent files from providing those features or characteristics; see the API documentation to read about provision handling and how to specify provisions as arguments to this action
	conary.build.packagepolicy.Provides-class.html
r.Requires	Mark a file requiring certain features or characteristics provided by a certain component (from the same or different package), or prevent the file from requiring that component; see the API documentation to read about requirement handling and how to specify requirements and exceptions as arguments to this action
	conary.build.packagepolicy.Requires-class.html
r.TagDescription	Mark a file being a tag description file, which provides execution information to Conary about a specific tag handler (see the description for the `r.TagHandler()` action); use this if you have tag description files which you are not installing in *%(tagdescriptiondir)s/* (Conary's default location of tag description files)
	conary.build.packagepolicy.TagDescription-class.html
r.TagHandler	Mark a file as being a tag handler, which is an executable file which performs an operation on some specific files each time those files are changed; use this if you have tag handlers which you are not installing in *%(taghandlerdir)s/* (Conary's default location of tag handlers)
	conary.build.packagepolicy.TagHandler-class.html
r.TagSpec	Apply tags defined by tag descriptions, both in the current system, and in the `%(destdir)s` of the package; also, specify any exceptions to these tags; see the `r.TagDescription()` and `r.TagHandler()` actions for more information about tags
	conary.build.packagepolicy.TagSpec-class.html
r.TestSuiteFiles	Copy extra files into the test directory; this would be used only if the `r.TestSuite()` build action is used
	conary.build.destdirpolicy.TestSuiteFiles-class.html
r.TestSuiteLinks	Create soft links in the test directory to files outside that directory; this would be used only if the `r.TestSuite()` build action is used

Action	Description and API Link
	conary.build.destdirpolicy.TestSuiteLinks-class.html
r.Transient	Mark a file as having transient contents, expected to change after installs and updates, but that is not already initial contents or a configuration file (by default Conary policy, or explicitly set with recipe actions `r.InitialContents` or `r.Config`); Conary overrides its default behavior, which would ignore modified files for updates, and completely overwrites these transient files during an update
	conary.build.packagepolicy.Transient-class.html
r.UtilizeGroup	Set a file to require the existance of a system group, even if the file is not owned by that group
	conary.build.packagepolicy.UtilizeGroup-class.html
r.UtilizeUser	Set a file to require the existance of a system user, even if the file is not owned by that user
	conary.build.packagepolicy.UtilizeUser-class.html

The second of two tables includes "pluggable" policy actions that are provided by the *conary-policy* package, which is installed on your build system when you install other development tools for packaging for Conary. The code provided provided by *conary-policy* is installed in a directory that Conary is configured to look at automatically for any pluggable actions. (To complete the URL for the links, start with: **http://cvs.rpath.com/conary-policy/**)

> **Note**
>
> You can also develop your own pluggable policy actions and add them to the same directory used by *conary-policy*. If you decide to explore this option, check out a copy of the *conary-policy* package to use as a reference.

Table A.10. Pluggable Policy Actions for Conary Recipes, from conary-policy

Action (Method Call)	Description and API Link
r.AutoDoc	Add documentation not otherwise installed
	autodoc.AutoDoc-class.html
r.BadFilenames	Require that filenames must not contain newlines; no exceptions allowed
	badpathnames.BadFilenames-class.html
r.BadInterpreterPaths	Enforce the use of absolute paths for interpreters

Action (Method Call)	Description and API Link
	badfilecontents.BadInterpreterPaths-class.html
r.CheckDesktopFiles	Warn about possible errors in desktop files, such as missing icon files; often used when you're also using the `r.Desktopfile()` build action
	badfilecontents.CheckDesktopFiles-class.html
r.CheckDestDir	Verify the absence of the destination directory (`%(destdir)s`) path in file paths and symbolic link contents; does not check inside files
	badpathnames.CheckDestDir-class.html
r.CheckSonames	Warn about any possible errors with regards to shared libraries (like `%(libdir)s/libexample.*`); *OR*add an `exceptions=` argument to override the warning about this
	libraries.CheckSonames-class.html
r.DanglingSymlinks	Do not allow dangling symbolic links, or symbolic links whose targets do not exist; *OR* add an `exceptions=` argument to allow exceptions, such as when your packages need to link to a file or directory that is not be managed by that package
	symlinks.DanglingSymlinks-class.html
r.EnforceCILBuildRequirements	Raise an error building the package if the package or its build requirements (`buildRequires`)do not meet Common Intermediate Language (CLI) requirements; *OR* add an `exceptions=` argument for all files, or specific files or components
	enforcebuildreqs.EnforceCILBuildRequirements-class.html
r.EnforceCMakeCacheBuildRequirements	Raise an error building the package if components listed in *CMakeCache.txt* files are not listed in the package's build requirements (`buildRequires`); *OR* add an `exceptions=` argument to disable the requirement for a given component
	enforcebuildreqs.EnforceCMakeCacheBuildRequirements-class.html
r.EnforceFlagBuildRequirements	Raise an error building the package if the package or its build requirements (`buildRequires`) do not include the files needed to define the flavor (flags) for which you're building the package; *this should never be called explicitly, and it takes no exceptions*

Action (Method Call)	Description and API Link
	enforcebuildreqs.EnforceFlagBuildRequirements-class.html
r.EnforceConfigLogBuildRequirements	Raise an error building the package if the package's build requirements (`buildRequires`) do not include items mentioned in `config.log` files; *OR* add an `exceptions=` argument to disable the requirement for a given component
	enforcebuildreqs.EnforceConfigLogBuildRequirements-class.html
r.EnforceJavaBuildRequirements	Raise an error building the package if the package or its build requirements (`buildRequires`) do not meet Java's runtime requirements; *OR* add an `exceptions=` argument to disable the requirement for all files, or specific files or components
	enforcebuildreqs.EnforceJavaBuildRequirements-class.html
r.EnforcePerlBuildRequirements	Raise an error building the package if the package or its build requirements (`buildRequires`) do not meet Perl's runtime requirements; *OR* add an `exceptions=` argument to disable the requirement for all files, or specific files or components
	enforcebuildreqs.EnforcePerlBuildRequirements-class.html
r.EnforcePythonBuildRequirements	Raise an error building the package if the package or its build requirements (`buildRequires`) do not meet Python's runtime requirements; *OR* add an `exceptions=` argument to disable the requirement for all files, or specific files or components
	enforcebuildreqs.EnforcePythonBuildRequirements-class.html
r.EnforceSonameBuildRequirements	Raise an error building the package if the package's build requirements (`buildRequires`) do not include the appropriate shared library dependencies; *OR* add an `exceptions=` argument to disable the requirement for all files, or specific files or components
	enforcebuildreqs.EnforceSonameBuildRequirements-class.html
r.ExecutableLibraries	Sets executable mode (`chmod +x`) on library files, and warn about doing so to prevent issues with `ldconfig`; *this should never be called explicitly in your recipe, but use the `r.SharedLibrary()` policy action instead*
	libraries.ExecutableLibraries-class.html

Action (Method Call)	Description and API Link
r.FilesForDirectories	Warn when encountering a file when a directory is expected, a scenario usually caused by bad `r.Install()` actions
	badpathnames.FilesForDirectories-class.html
r.FilesInMandir	Limit manual page entries to only locations where the `man` command can find them (`%(mandir)s`); *OR* add an `exceptions=` argument to disable this requirement for specified files
	badfilecontents.FilesInMandir-class.html
r.FixBuilddirSymlink	Remove the build directory (`%(builddir)s`) from symbolic links when appropriate
	symlinks.FixBuilddirSymlink-class.html
r.FixupManpagePaths	Repair manual page file paths caught by `r.FilesInMandir()` and other actions; *OR* add an `exceptions=` argument to disable this corrective action for specified files
	badfilecontents.FixupManpagePaths-class.html
r.FixupMultilibPaths	Repair and display a warning when files that should be installed in `lib64` are installed into `lib` instead; *OR* add an `exceptions=` argument to disable the corrective action for all or specified files
	libraries.FixupMultilibPaths-class.html
r.IgnoredSetupid	Display a warning when building the package if the user ID (UID) or group ID (GID) for a file or a directory is in conflict with the system's requirements; use `r.SetModes` build action to set these properly instead of using exceptions to this policy action
	permissions.IgnoredSetuid-class.html
r.ImproperlyShared	Verify that the shared data directory content (`%(datadir)s`) can be shared between system architectures; *OR* add an `exceptions=` argument to disable this requirement for all or specific files in the data directory
	badfilecontents.ImproperlyShared-class.html
r.NonBinariesInBindirs	Do not allow binary directory contents to include anything other than files that have their executable bits set (such as

Action (Method Call)	Description and API Link
	when using `r.SetModes()`); *OR* add an `exceptions=` argument to override this mode requirement for all or specified files
	badfilecontents.NonBinariesInBindirs-class.html
r.NonMultilibComponent	Enforce that a package should have multilib support so that both the 32-bit and 64-bit components can be installed for Python and Perl
	badpathnames.NonMultilibComponent-class.html
r.NonMultilibDirectories	Prevent using the *lib64* path for files that should install on 32-bit platforms
	badpathnames.NonMultilibDirectories-class.html
r.NonUTF8Filenames	Require UTF-8 filenames
	badpathnames.NonUTF8Filenames-class.html
r.NormalizeAppDefaults	Identify X application *defaults* files
	normalize.NormalizeAppDefaults-class.html
r.NormalizeCompression	Adjust compressed files for maximum compression; *OR* add an `exceptions=` argument to prevent this adjustment
	normalize.NormalizeCompression-class.html
r.NormalizeInfoPages	Use with an `exceptions=` argument to prevent Conary's default policy to compress *info* files (used with the `info` utility) and remove the *info* directory file
	normalize.NormalizeInfoPages-class.html
r.NormalizePythonInterpreterVersion	Verify that Python script files have a version-specific path to the Python interpreter; map directory paths to a particular Python version's interpreter if necessary; add exceptions to prevent this for specified files
	normalize.NormalizePythonInterpreterVersion-class.html
r.NormalizeInitscriptContents	Repair common errors in `init` scripts, and add dependencies if necessary; add exceptions to prevent this for specified files
	normalize.NormalizeInitscriptContents-class.html

Action (Method Call)	Description and API Link
r.NormalizeInitscriptLocation	Identifies `init` scripts and resolves issues in their location, moving them all to `%(initdir)s`); add exceptions to prevent this for specified files
	normalize.NormalizeInitscriptLocation-class.html
r.NormalizeInterpreterPaths	Rewrite interpreter paths in scripts, changing indirect calls through environment variables to direct calls to filesystem locations; add exceptions to prevent this for specified files
	normalize.NormalizeInterpreterPaths-class.html
r.NormalizeLibrarySymlinks	Execute the `ldconfig` command in each system library directory to prevent packaging unowned symbolic links; *this should never be called explicitly in your recipe, but use the* `r.SharedLibrary()` *policy action instead with its* `subtrees` *option*
	libraries.NormalizeLibrarySymlinks-class.html
r.NormalizeManPages	Force all manual pages to follow the expected Linux filesystem standards, including such actions as fixing symbolic links and changing modes on files; add exceptions to prevent this for specified files
	normalize.NormalizeManPages-class.html
r.NormalizePamConfig	Repair PAM configuration files by removing references to older module paths that no longer apply to modern PAM libraries; *no exceptions to this should be required*
	normalize.NormalizePamConfig-class.html
r.NormalizePkgConfig	Force all `pkg-config` files to be installed in `%(libdir)s`, making them multilib-safe; *exceptions are not recommended*
	pkgconfig.NormalizePkgConfig-class.html
r.ParseManifest	When packaging from RPMs, parse a file containing a manifest intended for RPM
	manifest.ParseManifest-class.html
r.PHPRequires	Determine PHP build requirements based on the PHP files, containing the correct version of the PHP interpreter; add exceptions to disable this requirement for all or specified files.

Action (Method Call)	Description and API Link
	phprequires.PHPRequires-class.html
r.PythonEggs	Verify there are no Python *.egg* files as part of the package, which are incompatible with Conary
	badpathnames.PythonEggs-class.html
r.ReadableDocs	Ensure files marked and managed as documentation are world-readable, with a mode set as in `chmod o+rw`
	permissions.ReadableDocs-class.html
r.RelativeSymlinks	Change all absolute symbolic links to be relative as appropriate for your package files
	symlinks.RelativeSymlinks-class.html
r.RemoveNonPackageFiles	Remove file types that should not be part of the package; add exceptions to allow any such files to be part of the package
	nonpackagefiles.RemoveNonPackageFiles-class.html
r.RequireChkconfig	Require all `init` script files to provide information for the `chkconfig` utility; add exceptions to override this requirement on any such files
	badfilecontents.RequireChkconfig-class.html
r.SharedLibrary	Mark system shared libraries so that `ldconfig` can run on them; see the API doc link for details on controlling this policy
	libraries.SharedLibrary-class.html
r.Strip	Remove debugging information from executables and libraries; add files or libraries as exceptions to keep debugging information in for future use
	strip.Strip-class.html
r.WarnWritable	Display a warning during the package build for any files which are unexpectedly set as group-writable or world-writable (equivalent to `chmod g+w` or `chmod o+w`); each `r.SetModes()` build action automatically creates exceptions
	permissions.WarnWriteable-class.html

Action (Method Call)	Description and API Link
r.WorldWriteableExecutables	Display a warning during the package build for any executable files which are world-writable (equivalent to `chmod o+wx`)
	permissions.WorldWriteableExecutables-class.html

A.8 Group Variables and Actions

Group recipes use a special part of the Conary API targeted to pulling together components, packages, and other groups so they can be installed and managed together. Groups allow you to identify specific versions of components that you know will work together, and to enforce version control on that group as a whole.

Groups recipes start something like this:

```
class GroupExample(GroupRecipe):
    name = 'group-example'
    version = '0.1'

    def setup(r):
        # Add items to the group here
```

From there, set additional variables and add recipe actions. You can set the following additional variables in a recipe inheriting from `GroupRecipe`:

- **depCheck** -- (default value: False) Set whether Conary should check for dependency closure in the group. If this is set to `True`, Conary will raise an error if there is no dependency closure within the group.
- **autoResolve** -- (default value: False) Control whether Conary should include the necessary components which automatically resolve dependencies in the group.
- **checkOnlyByDefaultDeps** -- (default value: True) Control whether Conary should check for dependencies in components which are installed by default only. If this is set to `False`, Conary will check dependencies in components which are *not* specified to be installed by default in addition to those which *will* be installed by default.
- **checkPathConflicts** -- (default value: True) Control whether Conary should check for path conflicts in each group to ensure the group can be installed without conflicts. Set this to `False` to disable checking for path conflicts.

After your variables are set as you need, add actions to the `setup()` method in the group. Choose from those listed in the following table. (To complete the URL for the links, start with:)

Table A.11. Group Actions

Action (Method Call)	Description and API Link
r.add	Add a component, package, or other group to your group
	conary.build.grouprecipe._GroupRecipe-class.html#add

Action (Method Call)	Description and API Link
r.addAll	Add all components, packages, or other groups that are directly contained within a specified group into your group (or a subgroup identified with `groupName=` if you're using `r.createGroup()`)
	conary.build.grouprecipe._GroupRecipe-class.html#addAll
r.addCopy	Create a copy of the named component, package, or other group, and add that copy to your group (or a subgroup identified with `groupName=` if you're using `r.createGroup()`)
	conary.build.grouprecipe._GroupRecipe-class.html#addCopy
r.addNewGroup	Add one newly created group to another newly created group to created a nesting structure with groups when you're using `r.createGroup()`
	conary.build.grouprecipe._GroupRecipe-class.html#addNewGroup
r.addResolveSource	Specify an alternate source for resolving group dependencies
	conary.build.grouprecipe._GroupRecipe-class.html#addResolveSource
r.copyComponents	Add components to your group by copying them from another group
	conary.build.grouprecipe._GroupRecipe-class.html#copyComponents
r.createGroup	Create a new group (subgroup to your group)
	conary.build.grouprecipe._GroupRecipe-class.html#createGroup
r.moveComponents	Add components to one group while removing them from another
	conary.build.grouprecipe._GroupRecipe-class.html#moveComponents
r.remove	Remove a component, package or other group from your group; typical when you're adding another group, but you need to remove certain items that are managed by that group
	conary.build.grouprecipe._GroupRecipe-class.html#remove
r.removeComponents	Indicate a component in your group that should not be installed by default when the group is installed
	conary.build.grouprecipe._GroupRecipe-class.html#removeComponents
r.removeItemsAlsoInGroup	Remove components or packages from a specified group (which you added to your your group) that are also added as components or packages in your own group

Action (Method Call)	Description and API Link
	conary.build.grouprecipe._GroupRecipe-class.html#removeItemsAlsoInGroup
r.removeItemsAlsoInNewGroup	Remove components or packages from a specified group (which you added to your your group) that are also added as components or packages in your own group
	conary.build.grouprecipe._GroupRecipe-class.html#removeItemsAlsoInNewGroup
r.Requires	Create a runtime requirement for the group, indicating that a specific component, package, or other group must be installed in order for your group to be installed
	conary.build.grouprecipe._GroupRecipe-class.html#Requires
r.replace	Replace a specific component, package, or other group with another one; typically used when you add a software group (such as a platform) to your group, but you have your own version of a package that should replace the one brought in by that software group (or platform)
	conary.build.grouprecipe._GroupRecipe-class.html#replace
r.setByDefault	Set components, packages, and other groups that should be added to your group by default
	conary.build.grouprecipe._GroupRecipe-class.html#setByDefault
r.setDefaultGroup	If you're using `r.createGroup()`, set the default group to which all components are added if a `groupName` argument is not provided to its corresponding add or replace action
	conary.build.grouprecipe._GroupRecipe-class.html#setDefaultGroup
r.setLabelPath	Specify on which labels to search for components, packages, and other groups to be included in your group
	conary.build.grouprecipe._GroupRecipe-class.html#setLabelPath
r.setSearchPath	Specify the search path on which to search for components, packages, and other groups to be included in your group; unlike `r.setLabelPath()`, this action can include both label and troveSpec arguments (a troveSpec is in the form <name>[=version][[flavor]] and is usually used to target a software group such as *group-dist* associated with a certain platform)
	conary.build.grouprecipe._GroupRecipe-class.html#setSearchPath

A.9 Legacy Recipe Actions

The legacy recipe actions in the following table are still a valid part of Conary code for compatibility with some older recipes, but their functionality has been superceded by features in other recipe actions. (To complete the URL for the links, start with: "http://cvs.rpath.com/conary-policy/")

Table A.12. Superceded Recipe Actions

Action	API Link
r.EtcConfig	*stubs.EtcConfig-class.html*
r.InstallBucket	*stubs.InstallBucket-class.html*
r.ObsoletePaths	*stubs.ObsoletePaths-class.html*

Appendix B. Recipe Types and Templates

When you need to create a new package, or just determine whether the rBuilder factory code is appropriate for your package, you can select an example recipe as a reference.

If you have a difficult time breaking down the recipe code and figuring out just what's going on, look at some of the basic recipe templates here to see how you could develop the recipe appropriate for your package.

Use these templates in conjunction with the following resources:

- Reference all the actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.
- Create and use repeatable recipe code to use across multiple packages using the information in *Factory Automation for Recipes* at *docs.rpath.com/conary*.

> **Completing the Build Requirements**
>
> Each template provided shows an empty `buildRequires` list. Conary already has a significant list of build requirements that it automatically uses, and it can report to you any other items you need to include in your own recipe. Place the empty `buildRequires` in your package recipe as a placeholder. Then, after you build the package for the first time, go back and add any build requirements as advised by Conary in the build messages.

B.1 Binary Executables

Use this section if the software you're packaging is an executable that is compiled and ready to launch on a target system.

If you're using rBuilder, you can create this type of package by using the Package Creator in its web interface. rBuilder combines its custom Conary factories with your uploaded source files to generate the recipe code you need. Conary and rBuilder combine their built-in intelligence to determine a lot about how the package should be assembled. rPath recommends that you take advantage of this built-in intelligence by continuing to use rBuilder's factories, and adding any supplemental and overriding code to the `preProcess` and `postProcess` sections of the recipe.

Without factories, you can create and maintain your package recipe to take advantage of the `binarypackage` super-class from Conary. Use the following as a template for starting your recipe:

```
# RECIPE TEMPLATE
# Package a binary/executable that is ready-to-run on the target system

loadSuperClass('binarypackage=conary.rpath.com@rpl:2')
class ExampleApp(BinaryPackageRecipe):
    name = 'exampleapp'
    version = '1.0'
    archive = 'http://www.example.com/exampleapp/%(name)s-%(version)s.tar.bz2'
    buildRequires = []
```

Replace the class name, package name, version, and the archive location as appropriate for an archive containing the executable files. The archive location can be just a file name if (you're checking in the file with the recipe), or a network-accessible location as shown in the template example.

If you need to change how the archive is unpacked and installed, or you need to include multiple archives, remove the "archive" line, and define an `unpack` method with the `r.addArchive` actions you need, instead:

```
# RECIPE TEMPLATE
# Package a binary/executable that is ready-to-run on the target system
# using an unpack method instead of the archive variable

loadSuperClass('binarypackage=conary.rpath.com@rpl:2')
class ExampleApp(BinaryPackageRecipe):
    name = 'exampleapp'
    version = '1.0'
    buildRequires = []

    def unpack(r):
        r.addArchive('http://www.example.com/exampleapp/%(name)s-%(version)s.tar.gz',
                     dir='/opt/%(name)s/', preserveOwnership=True)
        r.addArchive('http://www.example.com/exampleservice/%(name)s-%(version)-
service.tar.bz2',
                     dir='%(servicedir)s/%(name)s/)
```

To add policy actions to the recipe (overriding Conary's default policy), add on a `policy` method with the actions you need:

```
# RECIPE TEMPLATE
# Package a binary/executable that is ready-to-run on the target system
# adding on a policy method to override default Conary policy actions

loadSuperClass('binarypackage=conary.rpath.com@rpl:2')
class ExampleApp(BinaryPackageRecipe):
    name = 'exampleapp'
    version = '1.0'
    archive = 'http://www.example.com/exampleapp/%(name)s-%(version)s.tar.bz2'
    buildRequires = []

    def policy(r):
        r.Requires(exceptDeps='perl:.*')
        r.Requires('perl-Foo:runtime', '%(bindir)s/')
```

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.2 RPM Packages

Use this section if the software you're packaging is already provided as an RPM package. Choose whether you want to package the ready-to-install RPM using the (binary) RPM file, or you want to package from the source RPM (SRPM) so that you can make detailed customizations to how the software is installed.

B.2.1 Binary RPM Packages

If you're using rBuilder's Package Creator, you can create this package by uploading your RPM in the Web interface steps. rBuilder combines its custom Conary factories with your uploaded RPM file to generate the recipe code you need. Conary and rBuilder combine their built-in intelligence to determine a lot about how the package should be assembled. rPath recommends that you take advantage of this built-in intelligence by continuing to use rBuilder's factories, and adding any supplemental and overriding code to the `preProcess` and `postProcess` sections of the recipe.

Without factories, you could create and maintain your package recipe to take advantage of the `rpm-import` superclass associated with the platform for which you're developing. Use the following as a template for starting your recipe to package a binary RPM for Conary. There are currently no documents or recommended practices for using `rpm-import`, but you can view examples of its use by the RPM factories in each platform provided by rPath (such as in *factory-centos-rpm.recipe* in *factory-centos-rpm:source=centos.rpath.com@rpath:centos-5*).

B.2.2 Source RPM Packages

When you need to install your package differently from how the original RPM installs, you can also choose to package the software using the RPM source. The `rpmpackage` superclass from Conary is designed to parse information from a source RPM, apply any modifications you place in your recipe, and then package the software for Conary instead.

Use the following as a template for starting your recipe to assemble a Conary package from a source RPM:

```
# RECIPE TEMPLATE
# Package software from a source RPM

loadSuperClass('rpmpackage=conary.rpath.com@rpl:devel')
class ExampleApp(RPMPackageRecipe, AutoPackageRecipe):
    name = 'example'
    version = '3.1.0'
    rpmRelease = '4'
    rpmPatches = [ 'example-3.0.2.patch',
          'example-3.0.5.patch']
    externalArchive = 'http://www.example.com/%(name)s/%(name)s-%(version)s.tar.gz'
```

Replace the class name, package name, version, archive location, and RPM release as appropriate for your selected RPM. Each source and patch you add can be just a file name (if you're checking in those files with the recipe), or a network-accessible location as shown in the template example.

When you're using the `rpmpackage` superclass, you shouldn't need to define a `setup` or `unpack` method within your recipe, though you could add a `policy` method or similar to override some of Conary's default actions. For all the RPM-specific actions, you can set variables from the `rpmpackage` superclass by adding other lines like the `rpmRelease` line shown in the template.

You can use any of the following variables from the `rpmpackage` superclass:

- **externalArchive** -- Set this to pull patches or sources from the RPM, but want the upstream archive to come from a different location (expected input type is string)
- **tarballName** -- Use this if `rpmpackage` cannot determine the correct archive name contained within the RPM (expected input type is string)

- **distroVersion** -- Set this to the version of Red Hat Enterprise Linux (RHEL) or Fedora Core that should be used for fetching the RPM; defaults to "development" for Fedora Core and "4" for RHEL (expected input type is string)
- **rpmRelease** -- Add the RPM release string, which occurs after the dash and before the dot in the RPM name (expected input type is string)
- **isRHEL** -- RHEL sources are used of this is set to true; Fedora sources are used otherwise (expected input type is boolean)
- **rpmPatches** -- Add one or more patches to apply from the RPM (expected input type is a list of strings)
- **rpmSources** -- Add one or more sources to use from the RPM (expected input type is a list of strings)
- **rpmUpVer** -- Set the upstream package version if it should be different than the default, which is the recipe's version (rare; expected input type is string)
- **rpmName** -- Set the upstream package name if it should be different than the default, which is the recipe's name (rare; expected input type is string)

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.3 Java Applications

Use this section if the software you're packaging is compiled Java code that is ready to launch in any Java Runtime Environment (JRE).

Use the following as a template for starting your recipe to package your pre-compiled Java application:

```
# RECIPE TEMPLATE
# Package pre-compiled Java software
loadSuperClass('javapackage=conary.rpath.com@rpl:1')
class ExampleApp(JavaPackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []

    def upstreamUnpack(r):
        r.addArchive('http://www.example.com/%(example)s/%(example)s-%(version)s.tgz')
```

Replace the class name, package name, version, and archive location as appropriate for packaging the files for your Java application. Each source or archive you add can be just a file name (if you're checking in those files with the recipe), or a network-accessible location as shown in the template example.

> ### Packaging from JAR Files
>
> Future editions of this documentation will include information on packaging your Java application from its original source, provided in Java Archive (JAR) files.

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.4 C Source Code

Use this section if you want your package to start with C source code. Your package will have a two-part build process, first compiling the C code, and then packaging the resulting binaries.

If your C code compiles with the simple expected sequence "`./configure; make; make install`", you can spare extra lines in your package recipe and use the `AutoPackageRecipe` class from Conary:

```
# RECIPE TEMPLATE
# Package from C source code using the simple compiler sequence

class ExampleApp(AutoPackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []

    def unpack(r):
        r.addArchive('http://www.example.com/%(name)s-sources/%(name)s-%(version)s.tgz')
```

Replace the class name, package name, version, and archive location as appropriate for your package. Each source or archive you add can be:

- a file name (if you're checking in those files with the recipe)
- a network-accessible location as shown in the template example, or
- a version control checkout action from among Conary's source actions

For all C source code, you can use the `CPackageRecipe` class from Conary to package C code. Use this instead of `AutoPackageRecipe` when you need to make some additional adjustments before or during the compiling sequence:

```
# RECIPE TEMPLATE
# Package from C source code

class ExampleApp(CPackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []

    def setup(r):
        r.addArchive('http://www.example.com/%(name)s-sources/%(name)s-%(version)s.tgz')
        r.Make(makeName = 'Makefile-differentname')
        r.Run('%(builddir)s/pre-install.sh')
        r.MakeInstall()
        r.Install('final.exe', '%(datadir)s/%(name)s/')
```

Replace the class name, package name, version, and archive location as appropriate for your package. Each source or archive you add can be:

- a file name (if you're checking in those files with the recipe)
- a network-accessible location as shown in the template example, or
- a version control checkout action from among Conary's source actions

You can include `r.Configure` when building from source to include or exclude options when configuring. For more information on `r.Configure` see *Build Actions*.

> ### Beware of What Method to Use
>
> Note that the `AutoPackageRecipe` template is defines the `unpack` method in the recipe, and the `CPackageRecipe` defines the `setup` method in the recipe. You can always use the `setup` method inherited from Conary, but you may need to use `unpack` or some other method to take advantage of superclass-specific tools. Be sure to note which method is used in the reference recipe or template you use as a guide.

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.5 Python Source Code

Use this section if you want your package to start with Python source code. Because Conary is also built in Python, the tools needed to build your Python application into a Conary package are part of Conary's most basic recipe class, `PackageRecipe`.

Use the following template to start your recipe for packaging your Python application, taking advantage of Conary's Python setup tools:

```
# RECIPE TEMPLATE
# Package an application written in Python

class ExampleApp(PackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = [ 'python-setuptools:python' ]

    def setup(r):
        r.addArchive('http://www.example.com/%(name)s/%(name)s-%(version)s.tar.bz2')
        r.PythonSetup()
```

Replace the class name, package name, version, and archive location as appropriate for your package. Each source or archive you add can be just a file name (if you're checking in those files with the recipe), a network-accessible location as shown in the template example, or a version control checkout action from among Conary's source actions.

> ### Build Requirement Instead of Superclass
>
> The Python packaging template does not have a separate superclass adding recipe actions for you to use. Instead, you add `python-setuptools:python` to your build requirements so your build environment can run the `PythonSetup` recipe action.

If the application requires the use of the Python *site-packages* directory, adjust the recipe template as shown to load `python.recipe` and to set two important macros:

```
# RECIPE TEMPLATE
# Package an application written in Python that requires site-packages

loadInstalled('python')
class ExampleApp(PackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = [ 'python-setuptools:python' ]

    def setup(r):
        r.macros.pyver = Python.majversion
        r.macros.sitepkgs = '%(libdir)s/python%(pyver)s/site-packages'
        r.addArchive('http://www.example.com/%(name)s/%(name)s-%(version)s.tar.bz2')
        r.PythonSetup()
```

Again, replace the as appropriate for your package. Also, though, you can use the macro `%(sitepkgs)s` as appropriate in your recipe actions. Note that `%(libdir)s` on a 64-bit platform will be correct only if there is architecture-specific code in the Python library of your package.

Common Build Error

The following error might possibly occur when you build your package:

```
error: option --single-version-externally-managed not recognized
```

This error indicates that your `PythonSetup` build action will not work for your recipe. If this happens, replace the `PythonSetup` action with r.Run('./setup.py --<options>') where <options> are any package-specific options you need to pass to the Python setup. The following two lines work in many cases:

```
r.Run('python setup.py build')
r.Run('python setup.py install --root=%(destdir)s')
```

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.6 PHP Applications

If you want to package a PHP application, you can use Conary's basic recipe class `PackageRecipe` and just add the recipe actions needed to install the PHP files and set modes, ownership, and permissions on those files.

Use the following template to start this basic recipe for packaging a PHP application:

```
# RECIPE TEMPLATE
class ExampleApp(PackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []
```

```
    def setup(r):
        r.addArchive('http://www.example.com/%(name)s/'
                     '%(name)s-%(version)s.tar.bz2', dir='%(servicedir)s/%(name)s/')

        # Set modes and ownership on directories and files
        r.SetModes('%(servicedir)s/%(name)s/Sources', 0755)
        r.SetModes('%(servicedir)s/%(name)s/Plugins', 0755)
        r.SetModes('%(servicedir)s/%(name)s/index.php', 0644)
        r.Ownership('apache', 'root', '%(servicedir)s/%(name)s/Sources')
        r.Ownership('apache', 'root', '%(servicedir)s/%(name)s/Plugins')
        r.Ownership('apache', 'root', '%(servicedir)s/%(name)s/index.php')

        # Prevent removal of empty directory placeholders
        r.ExcludeDirectories(exceptions = '%(servicedir)s/%(name)s/Plugins')
```

You can also use the developer-contributed `PHPAppPackageRecipe` superclass from the *contrib* project at *contrib.rpath.org*. This superclass provides some recipe actions typical to installing PHP applications, and cuts down on the length of your recipe. This superclass (and other items in *contrib*) are not supported by rPath, but they are free and available for use in community projects. Use the following template if you want to use the `PHPAppPackageRecipe` superclass:

```
# RECIPE TEMPLATE
loadSuperClass('phpapppackage=contrib.rpath.org@rpl:devel')
class ExampleApp(PHPAppPackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []

    def unpack(r):
        r.extractArchive('http://www.example.com/%(name)s/'
                         '%(name)s-%(version)s.tar.bz2')
        r.CreateWriteable('%(approot)s/%(name)s.config')
        r.MakeWriteableDirs('%(approot)s/uploads')
        r.macros.dirconf = '    php_value memory_limit 16M'
```

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.7 Ruby Source Code

Use this section if you want your package to start with Ruby source code that contains a *setup.rb* file. There are no special superclasses dedicated to Ruby; you only need to use the `PackageRecipe` base class from Conary.

Use the following template to start this basic recipe for packaging a Ruby application:

```
# RECIPE TEMPLATE
# Package an application from Ruby source code that includes a setup.rb file

class ExampleApp(PackageRecipe):
    name = 'example'
    version = '1.0'
    buildRequires = []
```

```
    def setup(r):
        r.addArchive('http://www.example.com/example/'
                     '%(name)s-%(version)s.tar.bz2',
                      dir='/opt/%(name)s/')
        r.Run('ruby setup.rb config')
        r.Run('ruby setup.rb setup')
        r.Run('ruby setup.rb install --prefix="%(destdir)s")
```

Replace the class name, package name, version, and archive location as appropriate for your package. Each source or archive you add can be just a file name (if you're checking in those files with the recipe), a network-accessible location as shown in the template example, or a version control checkout action from among Conary's source actions.

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

The following example is a recipe for the Rake package:

```
# RECIPE TEMPLATE
# Package Rake

class Rake(PackageRecipe):
    name = 'rake'
    version = '0.7.3'

    buildRequires = [ 'ruby:runtime' ]

    def setup(r):
        r.addArchive('http://rubyforge.org/frs/download.php/19879/')

        # Install required libraries in site_ruby directory
        r.Install('lib/*', '%(libdir)s/ruby/site_ruby/')

        # Install 'binary' for rake (really just a wrapper script)
        r.Install('bin/*', '%(bindir)s/', mode=0755)
```

B.8 Perl Applications from CPAN

Use this section if you want your package to start with Perl source from Perl applications that are stored in *CPAN (search.cpan.org)*. The `cpanpackage` superclass from Conary is designed to pull the Perl code from CPAN and build and package it for Conary.

Use the following template to start this basic recipe for packaging a Perl application from CPAN:

```
# RECIPE TEMPLATE
# Package an application from Perl source provided from CPAN

loadSuperClass('cpanpackage=conary.rpath.com@rpl:2')
class ExampleApp(CPANPackageRecipe):
    # This package name should match the name of the package on CPAN
    name = 'perl-example'
    version = '1.0'
```

```
    # Either remove the author and server lines,
    # or change the value of "author" to be the CPAN author
    author = 'CPANAUTHORNAME'
    server = 'http://search.cpan.org/CPAN/'

    buildRequires = []
```

Replace the class name, package name, and version as appropriate for your package. Use the comments in the template as a guide for other actions. If the Perl module builds an accompanying C library (*.so*), also inherit from the `CPackageRecipe` superclass by changing the class declaration line to the following:

```
class ExampleApp(CPANPackageRecipe, CPackageRecipe)
```

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

You could use the following shell script to automate the creation of Conary recipes from a CPAN module. The script changes `IO::Socket::SSL` into `perl-IO-Socket-SSL` and then builds it. Note the Conary warning displayed with building the package, and add the `buildRequires` as necessary. Also, note that you can write a factory to do this, instead (see *Factory Automation for Conary Recipes* at *docs.rpath.com/conary*):

```
#!/bin/sh -x

# No rights reserved

#First make a new package
pkgname=`echo perl-$1 | sed -e s/::/-/g`
classname=`echo $1 | sed -e s/::///g`
version=$(perl -MCPAN -e "print stderr CPAN::Shell->expand(\"Module\",\"$1\")-
>cpan_version" 2>&1 1>/dev/null)
cvc newpkg $pkgname
cat >$pkgname/$pkgname.recipe <<EOF

loadSuperClass('cpanpackage=conary.rpath.com@rpl:devel')
class $classname(CPANPackageRecipe):
    name='$pkgname'
    version='$version'

    #In your recipe if you get an IO error, you may need to discover one or
    #both of the following
    #author='GBARR'
    #upstreamname='perl-Bundle-foo'

    buildRequires = [ ]

EOF

pushd $pkgname
cvc add $pkgname.recipe
cvc cook $pkgname.recipe
popd
```

Some applications using Perl contain private Perl modules. Conary discovers these are required by the application modifying the `@INC` array, but it has no way of discovering that the application also provides what it requires. Perl dependencies that are provided within the package should be explicitly removed with code similar to the following:

```
        r.Requires(exceptDeps='perl: (mod1|mod2|mod3|...)')
```

Conary does not expect that CPAN applications require this modification, but some applications that depend on CPAN modules may also require this modification for internal dependencies.

B.9 Custom Linux Kernel Packages

If you're building a system on a Linux platform, you can customize the Linux kernel as you need. This section assumes you have sufficient experience with Linux kernels to know how to use this information when packaging. If you are new to modifying a Linux kernel, consult other Linux resources to learn anything else you need to know.

First, start by shadowing the *kernel:source* from your platform. See *Copying and Customizing Existing Conary Packages* at *docs.rpath.com/conary* for more information about creating and maintaining shadows.

If you're using a platform other that rPath Linux, the kernel package is built from another package and not from the original Linux kernel source. If you want to modify the kernels from these other platforms you can do one of the following:

- Modify the shadowed recipe to tweak and repackage the kernel RPMs or debs from the platform.
- If you've already built RPMs or debs with your custom kernel, just replace the RPMs or debs in the kernel package's manifest file to point to your new packages.
- If you want to use the package to build a custom kernel from the Linux kernel source, replace all the kernel recipe code to use the `kernelpackage` superclass from Conary. Use the rPath Linux *kernel* package recipe as a guide to using this superclass. A possible template would be as follows; be sure to replace the kernel version as appropriate:

  ```
  loadSuperClass('kernelpackage=conary.rpath.com@rpl:devel')
  class Kernel(KernelPackageRecipe):

      name = 'kernel'
      version = '2.6.15.3'

      def unpack(r):
          # Add all your patches here.
  ```

If you're using rPath Linux as your platform, your package is already built from kernel source code. You can modify the *kernel.recipe* in a couple of ways:

- If *kernel:recipe* does not have an `unpack()` definition containing the customizations, create the `unpack()` definition for the recipe.
- If *kernel:recipe* has an existing `unpack()` definition, modify it to add your customizations.

The following example shows how a custom patch should appear when added to `unpack()`:

```
loadSuperClass('kernelpackage=conary.rpath.com@rpl:devel')
class Kernel(KernelPackageRecipe):

    name = 'kernel'
    version = '2.6.15.3'
```

```
def unpack(r):
    r.addPatch('my-custom-feature.patch')
```

No matter what platform you're using, if you choose to package your custom kernel from the Linux kernel source, you need to be aware of the kernel flavor specifications and architectures you're building for. For more information about reading flavors, see *Conary Versions and Flavors* at *docs.rpath.com/conary*. To see what flavor specifications are relevant to your kernel build, look at the flags throughout the (superclass) recipe code in *kernelpackage:source=conary.rpath.com@rpl:devel*. At the time of this writing, the following flags/flavor specifications were used in the superclass to affect the kernel build:

- **x86** and **x86_64** -- these are the same architecture flavors you would use in building any other package or group
- **smp** -- use *kernel.smp* or *~kernel.smp* enable SMP support; use *!kernel.smp* or *~!kernel.smp* to disable SMP support
- **pae** -- use *kernel.pae* or *~kernel.pae* enable PAE support; use *!kernel.pae* or *~!kernel.pae* to disable PAE support
- **numa** -- use *kernel.numa* or *~kernel.numa* enable NUMA support; use *!kernel.numa* or *~!kernel.numa* to disable NUMA support

Configure your build environment as appropriate so that when you run your build command, you can build your kernel for the flavors you need. If you're using rBuild for appliances in rBuilder, this means adjusting your product definition XML code to add relevant flavors, or using rMake or `cvc` instead while accounting for the needs of your product definition. If you're using rMake or `cvc`, this means making sure you use the appropriate flavor specifications in your *.rmakerc* or *.conaryrc* files and/or in the arguments of your `rmake` or `cvc` commands. See documentation for your selected build tools for more information about building with certain flavor specifications.

You can add drastically different options with extensive modifications to the *config.base* file in the shadow. However, for a more convenient solution, or to build a kernel specific to a single hardware platform, you could construct a new *.config* file, perhaps based on Conary.

To construct a custom *.config* file and add it to your kernel package, use the following process:

1. Shadow `kernel:source` from your platform, check out the package, and build it with `cvc cook` using the `--prep` option to simply prepare a build directory without completing the package build.
2. Change to the *builddir* directory, run `make menuconfig` (or `xconfig` or `gconfig` as appropriate), and configure the kernel.
3. Copy the custom *.config* file to the kernel package directory alongside the recipe, and rename it to *dotconfig*.
4. Replace all the methods ("def" sections) in your kernel's recipe with the `unpack()` and `configure()` definitions shown here:

```
def unpack(r):
    # Add any patches here as needed.
    #r.addPatch('')

    # This is for your special build. Specify dotconfig to
    # be your config, made with menuconfig and the arch you're
    # building for (either x86.i686 or x86_64.x86_64).
    r.addSource('dotconfig')
    r.macros.cfgver = 'x86.i686'
```

```
    def configure(r):
        r.Copy('dotconfig', '.config')
```

5. Build and test your kernel package, making tweaks where necessary.

Remember to rebuild any groups that contain this kernel package after you build the kernel. Make sure the group pulls in your custom *kernel* from your label rather than *kernel* from your platform's label.

B.10 GNOME Applications

Use this section if you want to package a GNOME desktop application. This is not typical for appliance products; it is more applicable to Linux distributions built with rBuilder and/or Conary. Use `GnomePackageRecipe` after loading the superclass `gnomepackage.recipe` from Conary.

Use the following template to start this basic recipe for packaging a GNOME desktop application:

```
loadRecipe('gnomepackage.recipe')
class ExampleApp(GnomePackageRecipe):

    name = 'example'
    version = '1.0'
    buildRequires = []
    extraConfig = '--disable-scrollkeeper'
```

Replace the class name, package name, and version as appropriate for your package. This recipe will automatically obtain the package from among the official GNOME applications. You can override the archive location to package a GNOME application that is not from GNOME's site by adding the following lines, replacing the archive URI as appropriate:

```
    def unpack(r):
        r.addArchive('http://www.example.com/%(name)s/')
```

Reference all the other actions you can add to your recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.11 Group Recipes

Use this section if you want to create a group of packages and components that should be installed and maintained together, not as separate pacakges. One type of group is the appliance group (such as *group-example-appliance*) that rBuilder creates to assemble all the software that should be included in your appliance. For that group's development, be sure to reference rBuilder documentation for developing your appliance group recipe.

You can also create a Conary group to put together smaller set of software that should always be managed together. You can create it in the same way as creating a new package, but the group must have the "group-" prefix in its name. A typical use case is locking in particular versions of interacting software that you have tested together.

Use the following template to start a group recipe:

```
class GroupExample(GroupRecipe):
```

```
    name = 'group-example'
    version = '0.1'

    # If you want the group build to automatically resolve any
    # dependencies, you can set autoResolve = True
    #
    autoResolve = True

    def setup(r):
        r.add('example:runtime', '/example.rpath.org@corp:example-4/4.0.1')
        r.add('mysql', '5.0.51a')
        r.add('php', '5.2.8')
```

Replace the class name, group name, and version as appropriate for your package, and adjust the variable values and add actions as needed. Groups can also contain other groups, which allows you to add this group to larger groups (like your appliance group recipe) if you want to do that. Be sure to build the group, and then test that it can install and manage the selected packages and components.

Remember these important facts about Conary groups and how they behave when they're installed on a Conary-based system:

- Packages in a group are installed and managed together. This "lock-in" feature is the reason to use groups. As such, the latest version of the group may not have the latest version of any of its packages.
- If you update an individual package on a system, then if package was previously installed and managed by a group, it is no longer managed by the group. Recommended practice is to update the group recipe and rebuild the group to include the desired package version; then, update the group on the target system so it can pull in your changes.

Reference all the other actions you can add to your group recipe in *Conary Recipe Actions, Macros, and Variables* at *docs.rpath.com/conary*.

B.12 Creating System Users and Groups with Info Packages

Use an *info-* package to create system users or groups (Linux groups, not Conary groups) on an underlying Linux system. This is typically necessary when you're packaging an application that needs to run as a specific user other than root. You can pair your application package with one or more *info-* packages as a dependency or in the same Conary group (such as in an appliance).

Create an *info-* package as you would any other package, and then use one of the templates here to start your package recipe.

Use the following template to start an *info-* package recipe used to create a new Linux system user:

```
class info_appuser(UserInfoRecipe):
    name = 'info-appuser'
    version = '1'

    def setup(r):
        r.User('appuser', 1001, group='examplegroup', groupid=1001,
               homedir='/home/appuser', shell='%(essentialbindir)s/bash',
               supplemental=['wheel','mysql'])
```

Replace the class name, package name, and version as appropriate, and replace the values in the User action to set up your Linux system user's information. You can remove or add arguments to the `User` action, too, as you need them. Use the following page in the Conary API for details about each argument you can use in the `User` action:

http://cvs.rpath.com/conary-docs/conary.build.build.User-class.html

Use the following template to start an *info-* package recipe used to create a new Linux system group:

```
class info_examplegroup(GroupInfoRecipe):
    name = 'info-examplegroup'
    version = '1'

    def setup(r):
        r.Group('examplegroup', 999)
```

Replace the class name, package name, and version as appropriate, and replace the values in the `Group` action to set up your Linux system group's information. You can remove or add arguments to the `Group` action, too, as you need them, and you can add a `SupplementalGroup` action to create supplemental groups. Use the following pages in the Conary API for details about each arguments you can use in these action:

http://cvs.rpath.com/conary-docs/conary.build.build.Group-class.html

http://cvs.rpath.com/conary-docs/conary.build.build.SupplementalGroup-class.html

> ### Avoiding User and Group Conflicts
>
> Be aware of the system users and groups on the Linux platform for which you're developing. For rPath Linux, there is an rPath UID/GID registry that includes all the users and groups associated with packages in rPath Linux. This is currently maintained at *rPath Linux:rPath UID Registry* at *wiki.rpath.com*.

Removing an *info-* package with `conary erase` does not delete the entry from */etc/passwd* and causes updates to an *info-* package to fail silently. Updates to a removed *info-* package such as changing the group information or salted password therefore fail to propagate to a running appliance. To arbitrarily override the initial settings in *info-* packages, you should instead use group scripts for the appliance.

B.13 Redirect Packages and Groups

Software applications change over time, and your packaging needs will change with them. Use this section when the maintenance path for your package or group has changed. In other words, you have a package or group that is no longer maintained where it is, and it needs to redirect to a different package or group for ongoing maintenance. Your package or group could redirect to:

- The same package or group, but on a different label
- A different package or group
- Multiple other packages or groups (common when breaking a larger package into smaller pieces for modularity)

Use the following steps to redirect your package or group:

1. Check out the package or group source and open its recipe in your preferred text editor.
2. Remove any loaded recipes or superclasses above the class declaration.
3. Change the inheritance for the class so it only inherits `RedirectRecipe`:

   ```
   class Example(RedirectRecipe):
   ```

4. Remove all variable assignments except name and version, and change the version value to `0` (zero):

   ```
   name = 'example'
   version = '0'
   ```

   If you're redirecting to multiple targets, add one new variable `allowMulipleTargets` set equal to `True`:

   ```
   name = 'example'
   version = '0'
   allowMultipleTargets = True
   ```

5. Remove all methods, and just define a single `setup` method:

   ```
   def setup(r):
   ```

6. Add redirects to your `setup` method as follows:
   - If you're redirecting to only one other package or group, add a single `addRedirect` action with two arguments: the target package name, and the label on which it resides. The following is in a redirect recipe redirecting to *perl-Mail-SpamAssassin* on the rPath Linux development label:

     ```
         def setup(r):
             r.addRedirect('perl-Mail-SpamAssassin',
      'conary.rpath.com@rpl:devel')
     ```

   - If you're redirecting to multiple targets, use multiple `addRedirect` actions. If the targets are on the same label, consider code like in the following example, which creates a list of targets in a Python array, and then uses a for loop to iterate over that array when adding the redirect action:

     ```
         def setup(r):
             target_list = [ 'example-server', 'example-client', 'example-
     common']
             for x in target_list:
                 r.addRedirect( x, 'example.rpath.org@corp:example-4')
     ```

7. Build your package or group as with any other package or group, and test to be sure it redirects to the appropriate targets when installed.

You can always undo your redirect by changing the recipe back to a normal package or group recipe.

B.14 Encapsulate an Existing RPM

If you're using rBuilder and an encapsulated platform such as Red Hat Enterprise Linux, you can create your own "capsule" packages that work in the same way as other capsule packages.

Use the following steps to package an existing RPM as a capsule package for RHEL:

1.
```
rbuild checkout --new foopkg
```

2.
```
cd foopkg
```

3. Edit the following example recipe:

```
class FoopkgRecipe(CapsuleRecipe):
    name = 'foopkg'
    version = '1.0'

    def setup(r):
        r.addCapsule('http://example.com/foopkg.rpm')
```

4.
```
rbuild build packages foopkg --no-recurse
```

If you are packaging for multiple architectures, edit the following example recipe that shows working with x86 and x86_64 in the same recipe:

```
class FoopkgRecipe(CapsuleRecipe):
    name = 'foopkg'
    version = '1.0'

    def setup(r):
        r.addCapsule('foopkg.x86_64.rpm', use=Arch.x86_64)
        r.addCapsule('foopkg.i586.rpm', use=Arch.x86)
```

An alternate approach is to use a factory, as shown in the following steps:

1.
```
rbuild checkout --new packagename --factory capsule-rpm=platformlabel
```

platformlabel should be the common platform label, for example `centos-5-common` and not `centos-5e`.

2.
```
cd pkgname
```

3. Copy the RPM(s) into this directory.
4. Add a recipe:

```
class OverrideRecipe(FactoryRecipeClass):
    def postprocess(r):
        pass
```

5.

```
ls -1 *.rpm > manifest
```

Note that the *ls -1* is dash-one.

6.
```
cvc add manifest --text
```

7.
```
rbuild build packages pkgname
```