Packaging and Delivering Software With the Image Packaging System in Oracle® Solaris 11.1

Part No: E21383–01
October 2012

121203@25097

Contents

Preface

Packaging and Delivering Software With the Image Packaging System in Oracle Solaris 11.1 describes how to use the Oracle Solaris Image Packaging System (IPS) feature to create software packages for the Oracle Solaris 11 operating system (OS).

Who Should Use This Book

This manual is for software developers who want to create packages that can be installed on the Oracle Solaris 11 OS and maintained using IPS. This book is also for developers and system administrators who want to better understand IPS and how the Oracle Solaris OS is packaged using IPS. Underlying IPS design concepts are discussed so that readers can more readily understand and use the more advanced features of IPS.

How This Book Is Organized

- Chapter 1, "IPS Design Goals, Concepts, and Terminology," outlines the basic design philosophy of IPS and its expression as software patterns.
- Chapter 2, "Packaging Software With IPS," gets you started constructing your own packages.
- Chapter 3, "Installing, Removing, and Updating Software Packages," describes how the IPS client works internally when installing, updating, and removing the software installed in an image.
- Chapter 4, "Specifying Package Dependencies," explains the different types of IPS dependencies and how they can be used to construct working software systems.
- Chapter 5, "Allowing Variations," explains how to provide different installation options to the end user.
- Chapter 6, "Modifying Package Manifests Programmatically," explains how package manifests can be machine edited to automatically annotate and check the manifests.
- Chapter 7, "Automating System Change as Part of Package Installation," explains how to use the Service Management Facility (SMF) to automatically handle any necessary system changes that should occur as a result of package installation.
- Chapter 8, "Advanced Topics For Package Updating," discusses renaming, merging, and splitting packages, moving package contents, delivering multiple implementations of an application, and sharing information across boot environments.

- Chapter 9, "Signing IPS Packages," describes IPS package signing and how developers and quality assurance organizations can sign either new packages or existing, already signed packages.
- Chapter 10, "Handling Non-Global Zones," describes how IPS handles zones and discusses those cases where packaging needs to account for non-global zones.
- Chapter 11, "Modifying Published Packages," describes how administrators can modify existing packages for local conditions.
- Appendix A, "Classifying Packages," shows package information classification scheme definitions.
- Appendix B, "How IPS Is Used To Package the Oracle Solaris OS," describes how Oracle uses IPS features to package the Oracle Solaris OS.

Related Documentation

- Chapter 2, "Managing Services (Overview)," in *Managing Services and Faults in Oracle Solaris 11.1* describes the Oracle Solaris Service Management Facility (SMF) feature
- *Copying and Creating Oracle Solaris 11.1 Package Repositories*
- *Adding and Updating Oracle Solaris 11.1 Software Packages*
- *Creating and Administering Oracle Solaris 11.1 Boot Environments* and the `beadm`(1M) man page
- *Installing Oracle Solaris 11.1 Systems*
- *Oracle Solaris Administration: Oracle Solaris Zones, Oracle Solaris 10 Zones, and Resource Management*
- *Oracle Solaris 11.1 Administration: ZFS File Systems*

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Description	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows UNIX system prompts and superuser prompts for shells that are included in the Oracle Solaris OS. In command examples, the shell prompt indicates whether the command should be executed by a regular user or a user with privileges.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

◆ ◆ ◆ CHAPTER 1

IPS Design Goals, Concepts, and Terminology

This chapter outlines the basic design philosophy of IPS and its expression as software patterns.

IPS Design Goals

IPS is designed to eliminate some long-standing issues with previous software distribution, installation, and maintenance mechanisms that have caused significant problems for Oracle Solaris customers, developers, maintainers, and ISVs.

Principle IPS design goals include:

Minimize downtime.
Minimize planned downtime by making software update possible while machines are in production.

Minimize unplanned downtime by supporting quick reboot to known working software configurations.

Automate installation and update.
Automate, as much as possible, the installation of new software and updates to existing software.

Reduce media requirement.
Resolve the difficulties with ever-increasing software size and limited distribution media space.

Verify correct software installation.
Ensure that it is possible to determine whether a package is correctly installed as defined by the author (publisher) of the package. Such a check should not be spoofable.

Enable easy virtualization.
Incorporate mechanisms to allow for the easy virtualization of Oracle Solaris at a variety of levels, in particular using zones.

Simplify upgrade.
Reduce the effort required to generate patches or upgrades for existing systems.

Enable easy package creation.
Enable other software publishers (ISVs and end-users themselves) to easily create and publish packages for Oracle Solaris.

These goals led to the following ideas:

Create boot environments as needed.
Leverage ZFS snapshot and clone facilities to dynamically create boot environments on an as-needed basis.

- Since Oracle Solaris 11 requires ZFS as the root file system, zone file systems need to be on ZFS as well.
- Users can create as many boot environments as desired.
- IPS can automatically create boot environments on an as-needed basis, either for backup purposes prior to modifying the running system, or for installation of a new version of the OS.

Unify installation, patch, and update.
Eliminate duplicated mechanisms and code used to install, patch, and update.

This idea results in several significant changes to the way Oracle Solaris is maintained, including the following important examples:

- All OS software updates and patching are done directly with IPS.
- Any time a new package is installed, it is already exactly at the correct version.

Minimize opportunities to install incorrectly.
The requirement for unspoofable verification of package installation has the following consequences:

- If a package needs to support installation in multiple ways, those ways must be specified by the developer so that the verification process can take this into account.
- Scripting is inherently unverifiable since the packaging system cannot determine the intent of the script writer. This, along with other issues discussed later, led to the elimination of scripting during packaging operations.
- A package cannot have any mechanism to edit its own manifest, since verification is then impossible.
- If the administrator wants to install a package in a manner incompatible with the original publisher's definition, the packaging system should enable the administrator to easily republish the package he wants to alter so that the scope of his changes is clear, not lost across upgrades, and can be verified in the same manner as the original package.

Provide software repositories.
The need to avoid size restrictions led to a software repository model, accessed using several different methods. Different repository sources can be composited to provide a complete set

of packages, and repositories can be distributed as a single file. In this manner, no single media is ever required to contain all the available software. To support disconnected or firewalled operations, tools are provided to copy and merge repositories.

Include metadata as part of the software package.
The desire to enable multiple (possibly competing) software publishers led to the decision to store all the packaging metadata in the packages themselves: No master database exists for information such as all packages and dependencies. A catalog of available packages from a software publisher is part of the repository for performance reasons, but the catalog can also be regenerated from the data contained in the packages.

Software Self-Assembly

Given the goals and ideas described above, IPS introduces the general concept of software self-assembly: Any collection of installed software on a system should be able to build itself into a working configuration when that system is booted, by the time the packaging operation completes, or at software runtime.

Software self-assembly eliminates the need for install-time scripting in IPS. The software is responsible for its own configuration, rather than relying on the packaging system to perform that configuration on behalf of the software. Software self-assembly also enables the packaging system to safely operate on alternate images, such as boot environments that are not currently booted, or offline zone roots. In addition, since the self-assembly is performed only on the running image, the package developer does not need to cope with cross-version or cross-architecture runtime contexts.

Some operating system image preparation must be done before boot, and IPS manages this transparently. Image preparation includes updating boot blocks, preparing a boot archive (ramdisk), and, on some architectures, managing the menu of boot choices.

Tools for Software Self-Assembly

The following IPS features and characteristics facilitate software self-assembly.

Atomic Software Objects

An *action* is the atomic unit of software delivery in IPS. Each action delivers a single software object. That software object can be a file system object such as a file, directory, or link, or a more complex software construct such as a user, group, or driver. In the SVR4 packaging system, these more complex action types are handled by using class action scripts. In IPS, no scripting is required.

Actions, grouped together into packages, can be installed, updated, and removed from both live images and offline images.

Actions are discussed in more detail in "Package Content: Actions" on page 20.

Configuration Composition

Rather than maintaining complex configuration files that require extensive scripting to update each configuration file during packaging operations, IPS encourages delivering fragments of configuration files. The packaged application can access those fragments directly when reading its configuration, or the fragments can be assembled into the complete configuration file before reading the file.

The Oracle Solaris 11 user attributes database is a good example of fragmented configuration files. The `/etc/user_attr` configuration file is used to configure extended attributes for roles and users on the system. In Oracle Solaris 11, the `/etc/user_attr` file is used for local changes only. Complete configuration is read from the separate files delivered into the `/etc/user_attr.d` directory. Multiple packages deliver fragments of the complete configuration. No scripting is needed when fragments are installed, removed, or updated.

This method of composing configuration files requires that the software is written with composition in mind, which is not always possible.

An alternative way to support composition is for a service to treat the configuration file as volatile, and reassemble the configuration file when fragments of the configuration are installed, removed, or updated. Typically, this assembly is performed by an SMF service. Assembly by an SMF service is discussed further in the next item.

Actuators and SMF Services

An *actuator* is a tag applied to any action delivered by the packaging system that causes a system change when that action is installed, removed, or updated. These changes are typically implemented as SMF services.

SMF services can configure software directly, or SMF services can construct configuration files using data delivered in the SMF manifest or sourced from files installed on the system.

SMF services have a rich syntax to express dependencies. Each service runs only when all of its dependencies have been met.

Any service can add itself as a dependency on the `svc:/milestone/self-assembly-complete:default` SMF milestone. Once the booting operating system has reached this milestone, all self-assembly operations should be completed.

A special type of zone called an *Immutable Zone* is a zone that can be configured to have restricted write access to portions of its file system. See the discussion of `file-mac-profile` in the `zonecfg`(1M) man page. To complete self-assembly in this type of zone, boot the zone read/write. After the `self-assembly-complete` SMF milestone, the zone is automatically booted to the required `file-mac-profile` setting.

Examples of Software Self-Assembly in Oracle Solaris

The following examples describe packages that are delivered as part of Oracle Solaris.

Apache Web Server

A good example of self-assembly is in the Oracle Solaris package for Apache Web Server: `pkg:/web/server/apache-22`. This package ships with a default `httpd.conf` file that has an `Include` directive that references `/etc/apache2/2.2/conf.d`:

```
Include /etc/apache2/2.2/conf.d/*.conf
```

Another package can deliver a new `.conf` file to that directory and use a `refresh_fmri` actuator to automatically refresh the Apache instance whenever the package that delivers this new `.conf` file is installed, updated, or removed. Refreshing the Apache instance causes the web server to rebuild its configuration.

```
file etc/apache2/2.2/conf.d/custom.conf path=etc/apache2/2.2/conf.d/custom.conf \
    owner=root group=bin mode=0644 refresh_fmri=svc:/network/http:apache22
```

See "Add Any Facets or Actuators That Are Needed" on page 39 and Chapter 7, "Automating System Change as Part of Package Installation," for information about how to use the `refresh_fmri` actuator.

Multiple Packages Delivering Configuration Fragments

Another example of self-assembly in the Oracle Solaris OS is shown in several packages that deliver content to the `/etc/security/exec_attr.d/` directory.

In earlier Oracle Solaris releases, an SMF service merged the files delivered in `exec_attr.d` into a single database, `/etc/security/exec_attr`. In the Oracle Solaris 11 OS, `libsecdb` reads the fragments in `exec_attr.d` directly, eliminating the need for a separate service to perform the merge.

Other directories containing fragments of configuration in `/etc/security` are handled in a similar way.

IPS Package Lifecycle

This section provides high-level descriptions of each state in the IPS package lifecycle. For best results, both package developers and system administrators should understand the various phases of the package lifecycle.

Create Packages can be created by anyone. IPS does not impose any particular software build system or directory hierarchy on package authors. For details about package creation, see Chapter 2, "Packaging Software With IPS." Aspects of package creation are discussed throughout the remaining chapters of this guide.

Publish Packages are published to an IPS repository, either to an HTTP location or to the file system. A published package can also be converted to a `.p5p` package archive file. To access software from an IPS repository, the repository can be added to the system using the `pkg set-publisher` command, or the repository can be accessed as a temporary source by using the `-g` option with `pkg` commands. Examples of package publication are shown in Chapter 2, "Packaging Software With IPS."

Install Packages can be installed on a system, either from an IPS repository accessed over `http://`, `https://`, or `file://` URLs, or from a `.p5p` package archive. Package installation is described in more detail in Chapter 3, "Installing, Removing, and Updating Software Packages."

Update Updated versions of packages might become available, either published to an IPS repository, or delivered as a new `.p5p` package archive. Installed packages can then be brought up to date, either individually, or as part of an entire system update.

Note that IPS does not use the same concept of "patching" that the SVR4 packaging system did. All changes to IPS packaged software are delivered by updated packages.

Package updates are performed in much the same way as package installations, but the packaging system is optimized to install only the changed portions delivered by an updated package. Package updating is described in more detail in Chapter 3, "Installing, Removing, and Updating Software Packages."

Rename During the life of a package, you might want to rename the package. A package might be renamed for organizational reasons or to refactor packages. Examples of package refactoring include combining several packages into a single package or breaking a single package into multiple smaller packages.

IPS gracefully handles content that moves between packages. IPS also allows old package names to persist on the system, automatically installing the new packages when a user asks to install a renamed package. Package renaming is described in more detail in Chapter 10.

Obsolete Eventually a package might reach the end of its life. A package publisher might decide that a package will no longer be supported, and that it will not have any more updates made available. IPS allows publishers to mark such packages as obsolete.

Obsolete packages can no longer be used as a target for most dependencies from other packages, and any packages upgraded to an obsolete version are automatically removed from the system. Package obsoletion is described in more detail in "Renaming, Merging and Splitting Packages" on page 81.

Remove	Finally, a package can be removed from the system if no other packages have dependencies on it. Package removal is described in more detail in Chapter 3, "Installing, Removing, and Updating Software Packages."

IPS Terminology and Components

This section defines IPS terms and describes IPS components.

Installable Image

IPS is designed to install packages in an image. An image is a directory tree, and can be mounted in a variety of locations as needed. An image is one of the following three types:

Full	In a full image, all dependencies are resolved within the image itself, and IPS maintains the dependencies in a consistent manner.
Zone	Non-global zone images are linked to a full image (the parent global zone image), but do not provide a complete system on their own. In a zone image, IPS maintains the non-global zone consistent with its global zone as defined by dependencies in the packages.
User	User images contain only relocatable packages.

In general, images are created or cloned by installers, `beadm`(1M), or `zonecfg`(1M), for example, rather than by `pkg image-create`.

Package Identifier: FMRI

Every IPS package is represented by a fault management resource identifier (FMRI) that consists of a publisher, a name, and a version, with the scheme `pkg`. In the following example package FMRI, `solaris` is the publisher, `system/library` is the package name, and `0.5.11,5.11-0.175.0.0.0.2.1:20111019T082311Z` is the version:

```
pkg://solaris/system/library@0.5.11,5.11-0.175.1.0.0.2.1:20120919T082311Z
```

FMRIs can be specified in abbreviated form if the resulting FMRI is still unique. The scheme, publisher, and version can be omitted. Leading components can be omitted from the package name.

- When the FMRI starts with `pkg://` or `//`, the first word following `//` must be the publisher name, and no components can be omitted from the package name. When no components are omitted from the package name, the package name is considered complete, or *rooted*.
- When the FMRI starts with `pkg:/` or `/`, the first word following the slash is the package name, and no components can be omitted from the package name. No publisher name can be present.
- When the version is omitted, the package generally resolves to the latest version of the package that can be installed.

Package Publisher

A publisher is an entity that develops and constructs packages. A publisher name, or prefix, identifies this source in a unique manner. Publisher names can include upper and lower case letters, numbers, hyphens, and periods: the same characters as a valid host name. Internet domain names or registered trademarks are good choices for publisher names, since these provide natural namespace partitioning.

Package clients combine all specified sources of packages for a given publisher when computing packaging solutions.

Package Name

Package names are hierarchical with an arbitrary number of components separated by forward slash (`/`) characters. Package name components must start with a letter or number, and can include underscores (`_`), hyphens (`-`), periods (`.`), and plus signs (`+`). Package name components are case sensitive.

Package names form a single namespace across publishers. Packages with the same name and version but different publishers are assumed to be interchangeable in terms of external dependencies and interfaces.

Leading components of package names can be omitted if the package name that is used is unique. For instance, `/driver/network/ethernet/e1000g` can be reduced to `network/ethernet/e1000g`, `ethernet/e1000g`, or even simply `e1000g`. When no components are omitted from the package name, the package name is considered complete, or *rooted*. If the packaging client complains about ambiguous package names, specify more components of the package name or specify the full, rooted name. Package names should be chosen to reduce possible ambiguities as much as possible.

If an FMRI contains a publisher name, then the full, rooted package name must be specified.

Scripts should refer to packages by their full, rooted names.

FMRIs can also be specified using an asterisk (*) to match any portion of a package name. Thus `/driver/*/e1000g` and `/dri*00g` both expand to `/driver/network/ethernet/e1000g`.

Package Version

A package version consists of four sequences of integer numbers, separated by punctuation. The elements in the first three sequences are separated by dots, and the sequences are arbitrarily long. Leading zeros in version elements are forbidden, to allow for unambiguous sorting by package version. For example, 01.1 and 1.01 are invalid version elements.

In the following example package version, the first sequence is `0.5.11`, the second sequence is `5.11`, the third sequence is `0.175.1.0.0.2.1`, and the fourth sequence is `20120919T082311Z`.

```
0.5.11,5.11-0.175.1.0.0.2.1:20120919T082311Z
```

Component version	The first sequence is the component version. For components that are developed as part of Oracle Solaris, this sequence represents the point in the release when this package last changed. For a component with its own development life cycle, this sequence is the dotted release number, such as 2.4.10.
Build version	The second sequence is the build version. This sequence, if present, must follow a comma. Oracle Solaris uses this sequence to denote the release of the OS for which the package was compiled.
Branch version	The third sequence is the branch version, providing vendor-specific information. This sequence, if present, must follow a hyphen. This sequence can contain a build number or provide some other information. This value can be incremented when the packaging metadata is changed, independently of the component. See "Oracle Solaris Package Versioning" on page 111 for a description of how the branch version fields are used in Oracle Solaris.
Time stamp	The fourth sequence is a time stamp. This sequence, if present, must follow a colon. This sequence represents the date and time the package was published in the GMT time zone. This sequence is automatically updated when the package is published

The package versions are ordered using left-to-right precedence: The number immediately after the `@` is the most significant part of the version space. The time stamp is the least significant part of the version space.

The `pkg.human-version` attribute can be used to hold a human-readable version string, however the versioning scheme described above must also be present. The human-readable version string is only used for display purposes, as documented in "Set Actions" on page 24.

By allowing arbitrary version lengths, IPS can accommodate a variety of different models for supporting software. For example, a package author can use the build or branch versions and assign one portion of the versioning scheme to security updates, another for paid versus unpaid support updates, another for minor bug fixes, or whatever information is needed.

A version can also be the token `latest`, which specifies the latest version known.

Appendix B, "How IPS Is Used To Package the Oracle Solaris OS," describes how Oracle Solaris implements versioning.

Package Content: Actions

Actions define the software that comprises a package; they define the data needed to create this software component. Package contents are expressed in a package manifest file as a set of actions.

Package manifests are largely created using programs. Package developers provide minimal information, and the manifest is completed using package development tools as described in Chapter 2, "Packaging Software With IPS."

Actions are expressed in the following form in package manifest files:

action_name attribute1=value1 attribute2=value2 ...

In the following example action, `dir` indicates this action specifies a directory. Attributes in the form *name*=*value* describe properties of that directory:

```
dir path=a/b/c group=sys mode=0755 owner=root
```

The following example shows an action that has data associated with it. In this `file` action, the second field, which has no *name*= prefix, is called the payload:

```
file 11dfc625cf4b266aaa9a77a73c23f5525220a0ef path=etc/release owner=root \
    group=sys mode=0444 chash=099953b6a315dc44f33bca742619c636cdac3ed6 \
    pkg.csize=139 pkg.size=189 variant.arch=i386
```

In this example, the payload is the SHA-1 hash of the file. This payload can alternatively appear as a regular attribute with the name `hash`, as shown in the following example. If both forms are present in the same action, they must have identical values.

```
file hash=11dfc625cf4b266aaa9a77a73c23f5525220a0ef path=etc/release owner=root \
    group=sys mode=0444 chash=099953b6a315dc44f33bca742619c636cdac3ed6 \
    pkg.csize=139 pkg.size=189 variant.arch=i386
```

Action metadata is freely extensible. Additional attributes can be added to actions as needed. Attribute names cannot include spaces, quotation marks, or equals signs (=). Attribute values can have all of those, although values with spaces must be enclosed in single or double quotation marks. Single quotation marks need not be escaped inside a string enclosed in double quotation marks, and double quotation marks need not be escaped inside a string enclosed in single quotation marks. A quotation mark can be prefixed with a backslash (`\`) to prevent terminating the quoted string. Backslashes can be escaped with backslashes. Custom attribute names should use a unique prefix to prevent accidental namespace overlap. See the discussion of publisher names in "Package Publisher" on page 18.

Multiple attributes with the same name can be present and are treated as unordered lists.

Most actions have a key attribute. The *key attribute* is the attribute that makes this action unique from all other actions in the image. For file system objects, the key attribute is the path for that object.

The following sections describe each IPS action type and the attributes that define these actions. The action types are detailed in the `pkg(5)` man page, and are repeated here for reference. Each section contains an example action as it would appear in a package manifest during package creation. Other attributes might be automatically added to the action during publication.

File Actions

The `file` action is by far the most common action. A `file` action represents an ordinary file. The `file` action references a payload, and has the following four standard attributes:

`path`	The file system path where the file is installed. This is the key attribute of a `file` action. The value of the `path` attribute is relative to the root of the image. Do not include the leading `/`.
`mode`	The access permissions of the file. The value of the `mode` attribute is simple permissions in numeric form, not ACLs.
`owner`	The name of the user that owns the file.
`group`	The name of the group that owns the file.

The payload is normally specified as a positional attribute: The payload is the first word after the action name and has no attribute name. In a published manifest, the payload value is the SHA-1 hash of the file contents. If the payload is present in a manifest that has not yet been published, it represents the path where the payload can be found, as explained in the `pkgsend(1)` man page. The named `hash` attribute must be used instead of the positional attribute if the payload value includes an equal symbol (`=`), double quotation mark (`"`), or space character. Both positional and `hash` attributes can be used in the same action, but the hashes must be identical.

A `file` action can also include the following attributes:

`preserve`	Specifies that the contents of the file should not be overwritten on upgrade if the contents are determined to have changed since the file was installed

or last upgraded. On initial installs, if an existing file is found, that existing file is salvaged (stored in `/var/pkg/lost+found`).

The `preserve` attribute can have one of the following values:

`renameold`	The existing file is renamed with the extension `.old`, and the new file is put in its place.
`renamenew`	The existing file is left alone, and the new file is installed with the extension `.new`.
`legacy`	This file is not installed for initial package installs. On upgrades, any existing file is renamed with the extension `.legacy`, and then the new file is put in its place.
`true`	The existing file is left alone, and the new file is not installed.

`overlay` Specifies whether the action allows other packages to deliver a file at the same location or whether it delivers a file intended to overlay another. This functionality is intended for use with configuration files that do not participate in any self-assembly (for example, `/etc/motd`) and that can be safely overwritten.

If `overlay` is not specified, multiple packages cannot deliver files to the same location.

The `overlay` attribute can have one of the following values:

`allow`	One other package is allowed to deliver a file to the same location. This value has no effect unless the `preserve` attribute is also set.
`true`	The file delivered by the action overwrites any other action that has specified `allow`.

Changes to the installed file are preserved based on the value of the `preserve` attribute of the overlaying file. On removal, the contents of the file are preserved if the action being overlaid is still installed, regardless of whether the `preserve` attribute was specified. Only one action can overlay another, and the `mode`, `owner`, and `group` attributes must match.

`original_name` This attribute is used to handle editable files moving from package to package, from place to place, or both. The value of this attribute is the name of the originating package, followed by a colon, followed by the original path to the file. Any file being deleted is recorded either with its package and path, or with the value of the `original_name` attribute if specified. Any

	editable file being installed that has the `original_name` attribute set uses the file of that name if it is deleted as part of the same packaging operation. Once this attribute is set, do not change its value, even if the package or file are repeatedly renamed. Keeping the same value permits upgrade to occur from all previous versions.
`release-note`	This attribute is used to indicate that this file contains release note text. The value of this attribute is a package FMRI. If the FMRI specifies a package name that is present in the original image and a version that is newer than the version of the package in the original image, this file will be part of the release notes. A special FMRI of `feature/pkg/self` refers to the containing package. If the version of `feature/pkg/self` is 0, this file will only be part of the release notes on initial installation.
`revert-tag`	This attribute is used to tag editable files that should be reverted as a set. Multiple `revert-tag` values can be specified The file reverts to its manifest-defined state when the `pkg revert` command is invoked with any of those tags specified. See the `pkg(1)` man page for information about the `revert` subcommand.

Specific types of files can have additional attributes. For ELF files, the following attributes are recognized:

`elfarch`	The architecture of the ELF file. This value is the output of `uname -p` on the architecture for which the file is built.
`elfbits`	This value is 32 or 64.
`elfhash`	This value is the hash of the ELF sections in the file that are mapped into memory when the binary is loaded. These are the only sections necessary to consider when determining whether the executable behavior of two binaries will differ.

An example `file` action is:

```
file path=usr/bin/pkg owner=root group=bin mode=0755
```

Directory Actions

The `dir` action is like the `file` action in that it represents a file system object, except that it represents a directory instead of an ordinary file. The `dir` action has the same four standard attributes as the `file` action (`path`, `owner`, `group`, and `mode`), and `path` is the key attribute.

Directories are reference counted in IPS. When the last package that either explicitly or implicitly references a directory no longer does so, that directory is removed. If that directory contains unpackaged file system objects, those items are moved into `/var/pkg/lost+found`.

Use the following attribute to move unpackaged contents into a new directory:

`salvage-from`	Names a directory of salvaged items. A directory with such an attribute inherits on creation the salvaged directory contents if they exist. For an example, see "Moving Unpackaged Contents on Directory Removal or Rename" on page 84.

During installation, `pkg(1)` checks that all instances of a given directory action on the system have the same `owner`, `group`, and `mode` attribute values. The `dir` action is not installed if conflicting values are found on the system or in other packages to be installed in the same operation.

An example of a `dir` action is:

```
dir path=usr/share/lib owner=root group=sys mode=0755
```

Link Actions

The `link` action represents a symbolic link. The `link` action has the following standard attributes:

`path`	The file system path where the symbolic link is installed. This is the key attribute for a `link` action.
`target`	The target of the symbolic link. The file system object to which the link resolves.

The `link` action also takes attributes that allow for multiple versions or implementations of a given piece of software to be installed on the system at the same time. Such links are mediated, and allow administrators to easily toggle which links point to which version or implementation as desired. These mediated links are discussed in "Delivering Multiple Implementations of an Application" on page 85.

An example of a `link` action is:

```
link path=usr/lib/libpython2.6.so target=libpython2.6.so.1.0
```

Hardlink Actions

The `hardlink` action represents a hard link. It has the same attributes as the `link` action, and `path` is also its key attribute

An example of a `hardlink` action is:

```
hardlink path=opt/myapplication/hardlink target=foo
```

Set Actions

The `set` action represents a package-level attribute, or metadata, such as the package description.

The following attributes are recognized:

`name`	The name of the attribute.
`value`	The value given to the attribute.

The `set` action can deliver any metadata the package author chooses. The following attribute names have specific meaning to the packaging system:

`pkg.fmri`	The name and version of the containing package.
`info.classification`	One or more tokens that a `pkg`(5) client can use to classify the package. The value should have a scheme (such as `org.opensolaris.category.2008` or `org.acm.class.1998`) and the actual classification (such as Applications/Games), separated by a colon (`:`). The scheme is used by the `packagemanager(1)` GUI. A set of `info.classification` values is provided in Appendix A, "Classifying Packages."
`pkg.summary`	A brief synopsis of the description. This value is shown at the end of each line of `pkg list -s` output, as well as in one line of the output of `pkg info`. This value should be no longer than 60 characters. This value should describe what the package is, and should not repeat the name or version of the package.
`pkg.description`	A detailed description of the contents and functionality of the package, typically a paragraph or so in length. This value should describe why someone might want to install this package.
`pkg.obsolete`	When `true`, the package is marked obsolete. An obsolete package can have no actions other than more `set` actions, and must not be marked renamed. Package obsoletion is covered in "Obsoleting Packages" on page 83.
`pkg.renamed`	When `true`, the package has been renamed. The package must include one or more `depend` actions as well, which point to the package versions to which this package has been renamed. A package cannot be marked both renamed and obsolete, but otherwise can have any number of `set` actions. Package renaming is covered in "Renaming, Merging and Splitting Packages" on page 81.
`pkg.human-version`	The version scheme used by IPS is strict and does not allow for letters or words in the `pkg.fmri` version field. If a commonly used human-readable version is available for a given package, that version can be set here. The value is displayed by IPS tools. This value is not used as a basis for version comparison and cannot be used in place of the `pkg.fmri` version.

Some additional informational attributes, as well as some used by Oracle Solaris are described in Appendix B, "How IPS Is Used To Package the Oracle Solaris OS."

An example of a set action is:

```
set name=pkg.summary value="Image Packaging System"
```

Driver Actions

The driver action represents a device driver. The driver action does not reference a payload. The driver files themselves must be installed as file actions. The following attributes are recognized. See add_drv(1M) for more information about these attribute values.

name	The name of the driver. This is usually, but not always, the file name of the driver binary. This is the key attribute of the driver action.
alias	An alias for the driver. A given driver can have more than one alias attribute. No special quoting rules are necessary.
class	A driver class. A given driver can have more than one class attribute.
perms	The file system permissions for the device nodes of the driver.
clone_perms	The file system permissions for the minor nodes of the clone driver for this driver.
policy	Additional security policy for the device. A given driver can have more than one policy attribute, but no minor device specification can be present in more than one attribute.
privs	Privileges used by the driver. A given driver can have more than one privs attribute.
devlink	An entry in /etc/devlink.tab. The value is the exact line to go into the file, with tabs denoted by \t. See the devlinks(1M) man page for more information. A given driver can have more than one devlink attribute.

An example of a driver action is:

```
driver name=vgatext \
    alias=pciclass,000100 \
    alias=pciclass,030000 \
    alias=pciclass,030001 \
    alias=pnpPNP,900 variant.arch=i386 variant.opensolaris.zone=global
```

Depend Actions

The `depend` action represents an inter-package dependency. A package can depend on another package because the first requires functionality in the second for the functionality in the first to work, or even to install. Dependencies are covered in Chapter 4, "Specifying Package Dependencies."

The following attributes are recognized:

`fmri`		The FMRI representing the target of the dependency. This is the key attribute of the `depend` action. The FMRI value must not include the publisher. The package name is assumed to be complete (that is, rooted), even if it does not begin with a forward slash (`/`). Dependencies of type `require-any` can have multiple `fmri` attributes. A version is optional on the `fmri` value, though for some types of dependencies, an FMRI with no version has no meaning. The FMRI value cannot use asterisks (`*`), and cannot use the `latest` token for a version.
`type`		The type of the dependency.
	`require`	The target package is required and must have a version equal to or greater than the version specified in the `fmri` attribute. If the version is not specified, any version satisfies the dependency. A package cannot be installed if any of its `require` dependencies cannot be satisfied.
	`optional`	The dependency target, if present, must be at the specified version level or greater.
	`exclude`	The containing package cannot be installed if the dependency target is present at the specified version level or greater. If no version is specified, the target package cannot be installed concurrently with the package specifying the dependency.
	`incorporate`	The dependency is optional, but the version of the target package is constrained. See Chapter 4, "Specifying Package Dependencies," for a discussion of constraints and freezing.
	`require-any`	Any one of multiple target packages as specified by multiple `fmri` attributes can satisfy the dependency, following the same rules as the `require` dependency type.
	`conditional`	The dependency target is required only if the package defined by the `predicate` attribute is present on the system.
	`origin`	Prior to installation of this package, the dependency target must, if present, be at the specified value or greater on the

	image to be modified. If the value of the `root-image` attribute is `true`, the target must be present on the image rooted at `/` in order to install this package.
`group`	The dependency target is required unless the package is on the image avoid list. Note that obsolete packages silently satisfy the `group` dependency. See the `avoid` subcommand in the `pkg(1)` man page for information about the image avoid list.
`parent`	The dependency is ignored if the image is not a child image, such as a zone. If the image is a child image, then the dependency target must be present in the parent image. The version matching for a `parent` dependency is the same as that used for `incorporate` dependencies.

`predicate`	The FMRI that represents the predicate for conditional dependencies.
`root-image`	Has an effect only for `origin` dependencies as mentioned above.

An example of a `depend` action is:

```
depend fmri=crypto/ca-certificates type=require
```

License Actions

The `license` action represents a license or other informational file associated with the package contents. A package can deliver licenses, disclaimers, or other guidance to the package installer through the `license` action.

The payload of the `license` action is delivered into the image metadata directory related to the package, and should only contain human-readable text data. The `license` action payload should not contain HTML or any other form of markup. Through attributes, `license` actions can indicate to clients that the related payload must be displayed or accepted. The method of display or acceptance is at the discretion of clients.

The following attributes are recognized:

`license`	Provides a meaningful description for the license to assist users in determining the contents without reading the license text itself. This is the key attribute of the `license` action. Some example values include: ■ ABC Co. Copyright Notice ■ ABC Co. Custom License ■ Common Development and Distribution License 1.0 (CDDL) ■ GNU General Public License 2.0 (GPL)

- GNU General Public License 2.0 (GPL) Only
- MIT License
- Mozilla Public License 1.1 (MPL)
- Simplified BSD License

Wherever possible, including the version of the license in the description is recommended as shown above. The `license` value must be unique within a package.

`must-accept`	When `true`, this license must be accepted by a user before the related package can be installed or updated. Omission of this attribute is equivalent to `false`. The method of acceptance (interactive or configuration-based, for example) is at the discretion of clients.
`must-display`	When `true`, the payload of the `license` action must be displayed by clients during packaging operations. Omission of this attribute is equivalent to `false`. This attribute should not be used for copyright notices, but only for actual licenses or other material that must be displayed during operations. The method of display is at the discretion of clients.

An example of a `license` action is:

```
license license="Apache v2.0"
```

Legacy Actions

The `legacy` action represents package data used by the legacy SVR4 packaging system. The attributes associated with the `legacy` action are added into the databases of the legacy SVR4 packaging system so that the tools querying those databases can operate as if the legacy package were actually installed. In particular, specifying the `legacy` action should cause the package named by the `pkg` attribute to satisfy SVR4 dependencies.

The following attributes are recognized. See the `pkginfo(4)` man page for description of the associated parameters.

`category`	The value for the CATEGORY parameter. The default value is `system`.
`desc`	The value for the DESC parameter.
`hotline`	The value for the HOTLINE parameter.
`name`	The value for the NAME parameter. The default value is `none provided`.
`pkg`	The abbreviation for the package being installed. The default value is the name from the FMRI of the package. This is the key attribute of the `legacy` action.
`vendor`	The value for the VENDOR parameter.

`version`	The value for the VERSION parameter. The default value is the version from the FMRI of the package.

An example of a `legacy` action is:

```
legacy pkg=SUNWcsu arch=i386 category=system \
    desc="core software for a specific instruction-set architecture" \
    hotline="Please contact your local service provider" \
    name="Core Solaris, (Usr)" vendor="Oracle Corporation" \
    version=11.11,REV=2009.11.11 variant.arch=i386
```

Signature Actions

Signature actions are used as part of the support for package signing in IPS. Signature actions are covered in detail in Chapter 9, "Signing IPS Packages."

User Actions

The `user` action defines a UNIX user as specified in the `/etc/passwd`, `/etc/shadow`, `/etc/group`, and `/etc/ftpd/ftpusers` files. Information from `user` actions is added to the appropriate files.

The following attributes are recognized:

`username`	The unique name of the user.
`password`	The encrypted password of the user. The default value is `*LK*`.
`uid`	The unique numeric ID of the user. The default value is the first free value under 100.
`group`	The name of the user's primary group. This name must be found in `/etc/group`.
`gcos-field`	The real name of the user, as represented in the GECOS field in `/etc/passwd`. The default value is the value of the `username` attribute.
`home-dir`	The user's home directory. The default value is `/`.
`login-shell`	The user's default shell. The default value is empty.
`group-list`	Secondary groups to which the user belongs. See the `group`(4) man page.
`ftpuser`	Can be set to `true` or `false`. The default value of `true` indicates that the user is permitted to login via FTP. See the `ftpusers`(4) man page.
`lastchg`	The number of days between January 1, 1970, and the date that the password was last modified. The default value is empty.

`min`	The minimum number of days required between password changes. This field must be set to 0 or above to enable password aging. The default value is empty.
`max`	The maximum number of days the password is valid. The default value is empty. See the `shadow(4)` man page.
`warn`	The number of days before `password` expires that the user is warned.
`inactive`	The number of days of inactivity allowed for the user. This is counted on a per-machine basis. The information about the last login is taken from the machine's `lastlog` file.
`expire`	An absolute date expressed as the number of days since the UNIX Epoch (January 1, 1970). When this number is reached, the login can no longer be used. For example, an expire value of 13514 specifies a login expiration of January 1, 2007.
`flag`	Set to empty.

A example of a `user` action is:

```
user gcos-field="pkg(5) server UID" group=pkg5srv uid=97 username=pkg5srv
```

Group Actions

The `group` action defines a UNIX group as specified in the `group(4)` file. No support is provided for group passwords. Groups defined with the `group` action initially have no user list. Users can be added with the `user` action.

The following attributes are recognized:

`groupname`	The value for the name of the group.
`gid`	The unique numeric ID of the group. The default value is the first free group under 100.

An example of a `group` action is:

```
group groupname=pkg5srv gid=97
```

Package Repository

A software repository contains packages for one or more publishers. Repositories can be configured for access in a variety of different ways: HTTP, HTTPS, file (on local storage or via NFS or SMB), and as a self-contained package archive file, usually with the `.p5p` extension.

Package archives allow for convenient distribution of IPS packages, and are discussed further in "Publish as a Package Archive" on page 43.

A repository accessed via HTTP or HTTPS has a server process, `pkg.depotd`, associated with it. See the `pkg.depotd`(1M) man page for more information. For an example, see "Retrieving Packages Using an HTTP Interface" in *Copying and Creating Oracle Solaris 11.1 Package Repositories*.

In the case of file repositories, the repository software runs as part of the accessing client. Repositories are created with the `pkgrepo` and `pkgrecv` commands as shown in *Copying and Creating Oracle Solaris 11.1 Package Repositories*.

CHAPTER 2

Packaging Software With IPS

This chapter gets you started constructing your own packages, including:

- Designing, creating, and publishing a new package
- Converting a SVR4 package to an IPS package

Designing a Package

Many of the criteria for good package development described in this section require you to make trade-offs. Satisfying all requirements equally is often difficult. The following criteria are presented in order of importance. However, this sequence is meant to serve as a flexible guide depending on your circumstances. Although each of these criteria is important, it is up to you to optimize these requirements to produce a good set of packages.

Select a package name.
Oracle Solaris uses a hierarchical naming strategy for IPS packages. Wherever possible, design your package names to fit into the same scheme. Try to keep the last part of your package name unique so that users can specify a short package name to commands such as `pkg install`.

Optimize for client-server configurations.
Consider the various patterns of software use (client and server) when laying out packages. Good package design divides the affected files to optimize installation of each configuration type. For example, for a network protocol implementation, the package user should be able to install the client without necessarily installing the server. If client and server share implementation components, create a base package that contains the shared bits.

Package by functional boundaries.
Packages should be self-contained and distinctly identified with a set of functionality. For example, a package that contains ZFS should contain all ZFS utilities and be limited to only ZFS binaries.

Packages should be organized from a user's point of view into functional units.

Package along license or royalty boundaries.
Put code that requires royalty payments due to contractual agreements or that has distinct software license terms in a dedicated package or group of packages. Do not disperse the code into more packages than necessary.

Avoid or manage overlap between packages.
Packages that overlap cannot be installed at the same time. An example of packages that overlap are packages that deliver different content to the same file system location. Since this error might not be caught until the user attempts to install the package, overlapping packages can provide a poor user experience. The `pkglint(1)` tool can help to detect this error during the package authoring process.

If the package content must differ, declare an `exclude` dependency so that IPS does not allow these packages to be installed together.

Correctly size packages.
A package represents a single unit of software, and is either installed or not installed. (See the discussion of facets in "Optional Software Components" on page 66 to understand how a package can deliver optional software components.) Packages that are always installed together should be combined. Since IPS downloads only changed files on update, even large packages update quickly if change is limited.

Creating and Publishing a Package

Packaging software with IPS is usually straightforward due to the amount of automation that is provided. Automation avoids repetitive tedium, which seems to be the principal cause of most packaging bugs.

Publication in IPS consists of the following steps:

1. Generate a package manifest.
2. Add necessary metadata to the generated manifest.
3. Evaluate dependencies.
4. Add any facets or actuators that are needed.
5. Verify the package.
6. Publish the package.
7. Test the package.

Generate a Package Manifest

The easiest way to get started is to organize the component files into the same directory structure that you want on the installed system.

Two ways to do this are:

- If the software you want to package is already in a tarball, unpack the tarball into a subdirectory. For many open source software packages that use the `autoconf` utility, setting the `DESTDIR` environment variable to point to the desired prototype area accomplishes this. The `autoconf` utility is available in the `pkg:/developer/build/autoconf` package.
- Use the `install` target in a Makefile.

Suppose your software consists of a binary, a library, and a man page, and you want to install this software in a directory under `/opt` named `mysoftware`. Create a directory in your build area under which your software appears in this layout. In the following example, this directory is named `proto`:

```
proto/opt/mysoftware/lib/mylib.so.1
proto/opt/mysoftware/bin/mycmd
proto/opt/mysoftware/man/man1/mycmd.1
```

Use the `pkgsend generate` command to generate a manifest for this `proto` area. Pipe the output package manifest through `pkgfmt` to make the manifest more readable. See the `pkgsend(1)` and `pkgfmt(1)` man pages for more information.

In the following example, the `proto` directory is in the current working directory:

```
$ pkgsend generate proto | pkgfmt > mypkg.p5m.1
```

The output `mypkg.p5m.1` file contains the following lines:

```
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

The path of the files to be packaged appears twice in the `file` action:

- The first word after the word `file` describes the location of the file in the `proto` area.
- The path in the `path=` attribute specifies the location where the file is to be installed.

This double entry enables you to modify the installation location without modifying the `proto` area. This capability can save significant time, for example if you repackage software that was designed for installation on a different operating system.

Notice that `pkgsend generate` has applied default values for directory owners and groups. In the case of `/opt`, the defaults are not correct. Delete that directory because it is delivered by other packages already on the system, and pkg(1) will not install the package if the attributes of /opt conflict with those already on the system. "Add Necessary Metadata to the Generated Manifest" on page 36 below shows a programmatic way to delete the unwanted directory.

If a file name contains an equal symbol (=), double quotation mark ("), or space character, pkgsend generates a hash attribute in the manifest, as shown in the following example:

```
$ mkdir -p proto/opt
$ touch proto/opt/my\ file1
$ touch proto/opt/"my file2"
$ touch proto/opt/my=file3
$ touch proto/opt/'my"file4'
$ pkgsend generate proto
dir group=bin mode=0755 owner=root path=opt
file group=bin hash=opt/my=file3 mode=0644 owner=root path=opt/my=file3
file group=bin hash="opt/my file2" mode=0644 owner=root path="opt/my file2"
file group=bin hash='opt/my"file4' mode=0644 owner=root path='opt/my"file4'
file group=bin hash="opt/my file1" mode=0644 owner=root path="opt/my file1"
```

When the package is published (see "Publish the Package" on page 42), the value of the hash attribute becomes the SHA-1 hash of the file contents, as noted in "File Actions" on page 21.

Add Necessary Metadata to the Generated Manifest

A package should define the following metadata. See "Set Actions" on page 24 for more information about these values and how to set these values.

`pkg.fmri`
: The name and version of the package as described in "Package Identifier: FMRI" on page 17. A description of Oracle Solaris versioning can be found in "Oracle Solaris Package Versioning" on page 111.

`pkg.description`
: A description of the contents of the package

`pkg.summary`
: A one-line synopsis of the description.

`variant.arch`
: Each architectures for which this package is suitable. If the entire package can be installed on any architecture, variant.arch can be omitted. Producing packages that have different components for different architectures is discussed in Chapter 5, "Allowing Variations."

`info.classification`
: A grouping scheme used by the packagemanager(1) GUI. The supported values are shown in Appendix A, "Classifying Packages." The example in this section specifies an arbitrary classification.

This example also adds a link action to /usr/share/man/index.d that points to the man directory under mysoftware. This link is discussed further in "Add Any Facets or Actuators That Are Needed" on page 39.

Rather than modifying the generated manifest directly, use pkgmogrify(1) to edit the generated manifest. See Chapter 6, "Modifying Package Manifests Programmatically," for a full description of using pkgmogrify to modify package manifests.

Create the following pkgmogrify input file to specify the changes to be made to the manifest. Name this file mypkg.mog. In this example, a macro is used to define the architecture, and regular expression matching is used to delete the /opt directory from the manifest.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description value="This is a full description of \
all the interesting attributes of this example package."
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
<transform dir path=opt$->drop>
```

Run pkgmogrify on the mypkg.p5m.1 manifest with the mypkg.mog changes:

```
$ pkgmogrify -DARCH=‘uname -p‘ mypkg.p5m.1 mypkg.mog | pkgfmt > mypkg.p5m.2
```

The output mypkg.p5m.2 file has the following content. The dir action for path=opt has been removed, and the metadata and link contents from mypkg.mog have been added to the original mypkg.p5m.1 contents.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
```

Evaluate Dependencies

Use the `pkgdepend(1)` command to automatically generate dependencies for the package. The generated `depend` actions are defined in "Depend Actions" on page 27 and discussed further in Chapter 4, "Specifying Package Dependencies."

Dependency generation is composed of two separate steps:

1. Dependency generation. Determine the files on which the software depends. Use the `pkgdepend generate` command.
2. Dependency resolution. Determine the packages that contain those files on which the software depends. Use the `pkgdepend resolve` command.

Generate Package Dependencies

In the following command, the `-m` option causes `pkgdepend` to include the entire manifest in its output. The `-d` option passes the `proto` directory to the command.

```
$ pkgdepend generate -md proto mypkg.p5m.2 | pkgfmt > mypkg.p5m.3
```

The output `mypkg.p5m.3` file has the following content. The `pkgdepend` utility added notations about a dependency on `libc.so.1` by both `mylib.so.1` and `mycmd`. The internal dependency between `mycmd` and `mylib.so.1` is silently omitted.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/bin/mycmd \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=opt/mysoftware/lib pkg.debug.depend.path=usr/lib
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/lib/mylib.so.1 \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=usr/lib
```

Resolve Package Dependencies

To resolve dependencies, pkgdepend examines the packages currently installed in the image used for building the software. By default, pkgdepend puts its output in mypkg.p5m.3.res. This step takes a while to run since it loads a large amount of information about the system on which it is running. The pkgdepend utility can resolve many packages at once if you want to amortize this time over all packages. Running pkgdepend on one package at a time is not time efficient.

```
$ pkgdepend resolve -m mypkg.p5m.3
```

When this completes, the output mypkg.p5m.3.res file contains the following content. The pkgdepend utility has converted the notation about the file dependency on libc.so.1 to a package dependency on pkg:/system/library, which delivers that file.

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is an example package"
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package."
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0644
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
depend fmri=pkg:/system/library@0.5.11-0.175.1.0.0.21.0 type=require
```

You should use pkgdepend to generate dependencies, rather than declaring depend actions manually. Manual dependencies can become incorrect or unnecessary as the package contents change over time. For example, when a file that an application depends on gets moved to a different package, any manually declared dependencies on the previous package would then be incorrect for that dependency.

Some manually declared dependencies might be necessary if pkgdepend is unable to determine dependencies completely. In such a case, you should add explanatory comments to the manifest.

Add Any Facets or Actuators That Are Needed

Facets and actuators are discussed in more detail in Chapter 5, "Allowing Variations," and Chapter 7, "Automating System Change as Part of Package Installation." A *facet* denotes an action that is not required but can be optionally installed. An *actuator* specifies system changes that must occur when the associated action is installed, updated, or removed

This example package delivers a man page in opt/mysoftware/man/man1. This section shows how to add a facet to indicate that man pages are optional. The user could choose to install all of the package except the man page. (If the user sets the facet to false, no man pages are installed from any package if their file actions are tagged with that facet.)

To include the man page in the index, the svc:/application/man-index:default SMF service must be restarted when the package is installed. This section shows how to add the restart_fmri actuator to perform that task. The man-index service looks in /usr/share/man/index.d for symbolic links to directories that contain man pages, adding the target of each link to the list of directories it scans. To include the man page in the index, this example package includes a link from /usr/share/man/index.d/mysoftware to /opt/mysoftware/man. Including this link and this actuator is a good example of the self-assembly discussed in "Software Self-Assembly" on page 13 and used throughout the packaging of the Oracle Solaris OS.

A set of pkgmogrify transforms that you can use are available in /usr/share/pkg/transforms. These transforms are used to package the Oracle Solaris OS, and are discussed in more detail in Chapter 6, "Modifying Package Manifests Programmatically."

The file /usr/share/pkg/transforms/documentation contains transforms similar to the transforms needed in this example to set the man page facet and restart the man-index service. Since this example delivers the man page to /opt, the documentation transforms must be modified as shown below. These modified transforms include the regular expression opt/.+/man(/.+)? which matches all paths beneath opt that contain a man subdirectory. Save the following modified transforms to /tmp/doc-transform:

```
<transform dir file link hardlink path=opt/.+/man(/.+)? -> \
    default facet.doc.man true>
<transform file path=opt/.+/man(/.+)? -> \
    add restart_fmri svc:/application/man-index:default>
```

Use the following command to apply these transforms to the manifest:

```
$ pkgmogrify mypkg.p5m.3.res /tmp/doc-transform | pkgfmt > mypkg.p5m.4.res
```

The input mypkg.p5m.3.res manifest contains the following three man-page-related actions:

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644
```

After the transforms are applied, the output mypkg.p5m.4.res manifest contains the following modified actions:

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755 facet.doc.man=true
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755 \
    facet.doc.man=true
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644 \
    restart_fmri=svc:/application/man-index:default facet.doc.man=true
```

For efficiency, these transforms could have been added when metadata was originally added, before evaluating dependencies.

Verify the Package

The last step before publication is to run `pkglint(1)` on the manifest to find errors that can be identified before publication and testing. Some of the errors that `pkglint` can find would also be found either at publication time or when a user attempts to install the package, but of course you want to identify errors as early as possible in the package authoring process.

Examples of errors that `pkglint` reports include:

- Delivering files already owned by another package.
- Difference in metadata for shared, reference-counted actions such as directories. An example of this error is discussed at the end of "Generate a Package Manifest" on page 34.

You can run `pkglint` in one of the following modes:

- Directly on the package manifest. This mode is usually sufficient to quickly check the validity of your manifests.
- On the package manifest, also referencing a package repository. Use this mode at least once before publication to a repository.

  By referencing a repository, `pkglint` can perform additional checks to ensure that the package interacts well with other packages in that repository.

Use the `pkglint -L` command to show the full list of checks that `pkglint` performs. Detailed information about how to enable, disable, and bypass particular checks is given in the `pkglint(1)` man page. The man page also details how to extend `pkglint` to run additional checks.

The following output shows problems with the example manifest:

```
$ pkglint mypkg.p5m.4.res
Lint engine setup...
Starting lint run...
WARNING pkglint.action005.1       obsolete dependency check skipped: unable
to find dependency pkg:/system/library@0.5.11-0.175.1.0.0.21.0 for
pkg:/mypkg@1.0,5.11-0
```

This warning is acceptable for this example. The `pkglint.action005.1` warning says that `pkglint` could not find a package called `pkg:/system/library@0.5.11-0.175.1.0.0.21.0`, on which this example package depends. The dependency package is in a package repository and could not be found since `pkglint` was called with only the manifest file as an argument.

In the following command, the `-r` option references a repository that contains the dependency package. The `-c` option specifies a local directory used for caching package metadata from the lint and reference repositories:

```
$ pkglint -c ./solaris-reference -r http://pkg.oracle.com/solaris/release mypkg.p5m.4.res
```

Publish the Package

IPS provides three different ways to deliver a package:

- Publish to a local file-based repository.
- Publish to a remote HTTP-based repository.
- Convert to a `.p5p` package archive.

Generally, publishing to a file-based repository is sufficient to test a package.

If the package must be transferred to other machines that cannot access the package repositories, converting one or more packages to a package archive can be convenient.

The package can also be published directly to an HTTP repository, hosted on a machine with a read/write instance of the `svc:/application/pkg/server` service, which in turn runs `pkg.depotd`(1M).

Publishing to an HTTP repository is not generally recommended since there are no authorization or authentication checks on the incoming package when publishing over HTTP. Publishing to HTTP repositories can be convenient on secure networks or when testing the same package across several machines if NFS or SMB access to the file repository is not possible.

Installing packages over HTTP or HTTPS is fine.

Publish to a Local File Repository

Use the `pkgrepo`(1) command to create and manage repositories. Choose a location on your system, create a repository, then set the default publisher for that repository:

```
$ pkgrepo create my-repository
$ pkgrepo -s my-repository set publisher/prefix=mypublisher
$ ls my-repository
pkg5.repository
```

Use the `pkgsend`(1) command to publish the example package, and then use `pkgrepo` to examine the repository:

```
$ pkgsend -s my-repository publish -d proto mypkg.p5m.4.res
pkg://mypublisher/mypkg@1.0,5.11-0:20120331T034425Z
PUBLISHED
$ pkgrepo -s my-repository info
PUBLISHER   PACKAGES STATUS           UPDATED
mypublisher 1        online           2012-03-31T03:44:25.235964Z
```

The file repository can then be served over HTTP or HTTPS using `pkg.depotd` if required.

Publish as a Package Archive

Package archives enable you to distribute groups of packages in a single file. Use the `pkgrecv(1)` command to create package archives from package repositories, or to create package repositories from package archives.

Package archives can be easily downloaded from an existing web site, copied to a USB key, or burned to a DVD for installation in cases where a package repository is not available.

The following command creates a package archive from the simple repository created in the previous section:

```
$ pkgrecv -s my-repository -a -d myarchive.p5p mypkg
Retrieving packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.7/0.7 17.9k/s

ARCHIVE                                              FILES   STORE (MB)
myarchive.p5p                                        14/14      0.7/0.7
```

Using Package Repositories and Archives

Use the `pkgrepo` command to list the newest available packages from a repository or archive:

```
$ pkgrepo -s my-repository list '*@latest'
PUBLISHER   NAME                                          O VERSION
mypublisher mypkg                                           1.0,5.11-0:20120331T034425Z
$ pkgrepo -s myarchive.p5p list '*@latest'
PUBLISHER   NAME                                          O VERSION
mypublisher mypkg                                           1.0,5.11-0:20120331T034425Z
```

This output can be useful for constructing scripts to create archives with the latest versions of all packages from a given repository.

Temporary repositories or package archives can be provided to `pkg install` and other `pkg` operations by using the `-g` option. Such temporary repositories and archives cannot be used on systems with child or parent images (for example, systems with non-global zones) since the system repository does not get temporarily configured with that publisher information. Non-global zones have a child/parent relationship with the global zone. Package archives can be set as sources of local publishers in non-global zones, however.

Test the Package

The final step in package development is to test whether the published package has been packaged properly.

To test installation without requiring root privilege, assign the test user the Software Installation profile. Use the `-P` option of the `usermod` command to assign the test user the Software Installation profile.

Note – If this image has child images (non-global zones) installed, you cannot use the `-g` option with the `pkg install` command to test installation of this package. You must configure the `my-repository` repository in the image.

Add the publisher in the `my-repository` repository to the configured publishers in this image:

```
$ pfexec pkg set-publisher -p my-repository
pkg set-publisher:
  Added publisher(s): mypublisher
```

You can use the `pkg install -nv` command to see what the install command will do without making any changes. The following command actually installs the package:

```
$ pfexec pkg install mypkg
           Packages to install:  1
       Create boot environment: No
Create backup boot environment: No
            Services to change:  1

DOWNLOAD                                  PKGS       FILES   XFER (MB)   SPEED
Completed                                  1/1         3/3     0.0/0.0    0B/s

PHASE                                          ITEMS
Installing new actions                         15/15
Updating package state database                 Done
Updating image state                            Done
Creating fast lookup database                   Done
Reading search index                            Done
Updating search index                            1/1
```

Examine the software that was delivered on the system:

```
$ find /opt/mysoftware/
/opt/mysoftware/
/opt/mysoftware/bin
/opt/mysoftware/bin/mycmd
/opt/mysoftware/lib
/opt/mysoftware/lib/mylib.so.1
/opt/mysoftware/man
/opt/mysoftware/man/man-index
/opt/mysoftware/man/man-index/term.doc
/opt/mysoftware/man/man-index/.index-cache
/opt/mysoftware/man/man-index/term.dic
/opt/mysoftware/man/man-index/term.req
/opt/mysoftware/man/man-index/term.pos
/opt/mysoftware/man/man1
/opt/mysoftware/man/man1/mycmd.1
```

In addition to the binaries and man page, the system has also generated the man page indexes as a result of the actuator restarting the `man-index` service.

The `pkg info` command shows the metadata that was added to the package:

```
$ pkg info mypkg
          Name: mypkg
       Summary: This is an example package
   Description: This is a full description of all the interesting attributes of
                this example package.
      Category: Applications/Accessories
         State: Installed
     Publisher: mypublisher
       Version: 1.0
 Build Release: 5.11
        Branch: 0
Packaging Date: March 31, 2012 03:44:25 AM
          Size: 0.00 B
          FMRI: pkg://mypublisher/mypkg@1.0,5.11-0:20120331T034425Z
```

The `pkg search` command returns hits when querying for files that are delivered by `mypkg`:

```
$ pkg search -l mycmd.1
INDEX      ACTION VALUE                           PACKAGE
basename   file   opt/mysoftware/man/man1/mycmd.1 pkg:/mypkg@1.0-0
```

Converting SVR4 Packages To IPS Packages

This section shows an example of converting a SVR4 package to an IPS package and highlights areas that might need special attention.

To convert a SVR4 package to an IPS package, follow the same steps described in above in this chapter for packaging any software in IPS. Most of these steps are the same for conversion from SVR4 to IPS packages and are not explained again in this section. This section describes the steps that are different when converting a package rather than creating a new package.

Generate an IPS Package Manifest from a SVR4 Package

The *source* argument of the `pkgsend generate` command can be a SVR4 package. See the `pkgsend(1)` man page for a complete list of supported sources. When *source* is a SVR4 package, `pkgsend generate` uses the `pkgmap(4)` file in that SVR4 package, rather than the directory inside the package that contains the files delivered.

While scanning the `prototype` file, the `pkgsend` utility also looks for entries that could cause problems when converting the package to IPS. The `pkgsend` utility reports those problems and prints the generated manifest.

The example SVR4 package used in this section has the following pkginfo(4) file:

```
VENDOR=My Software Inc.
HOTLINE=Please contact your local service provider
PKG=MSFTmypkg
ARCH=i386
DESC=A sample SVR4 package of My Sample Package
CATEGORY=system
NAME=My Sample Package
BASEDIR=/
VERSION=11.11,REV=2011.10.17.14.08
CLASSES=none manpage
PSTAMP=linn20111017132525
MSFT_DATA=Some extra package metadata
```

The example SVR4 package used in this section has the following corresponding prototype(4) file:

```
i pkginfo
i copyright
i postinstall
d none opt 0755 root bin
d none opt/mysoftware 0755 root bin
d none opt/mysoftware/lib 0755 root bin
f none opt/mysoftware/lib/mylib.so.1 0644 root bin
d none opt/mysoftware/bin 0755 root bin
f none opt/mysoftware/bin/mycmd 0755 root bin
d none opt/mysoftware/man 0755 root bin
d none opt/mysoftware/man/man1 0755 root bin
f none opt/mysoftware/man/man1/mycmd.1 0644 root bin
```

Running the `pkgsend generate` command on the SVR4 package built using these files generates the following IPS manifest:

```
$ pkgsend generate ./MSFTmypkg | pkgfmt
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall

set name=pkg.summary value="My Sample Package"
set name=pkg.description value="A sample SVR4 package of My Sample Package"
set name=pkg.send.convert.msft-data value="Some extra package metadata"
dir path=opt owner=root group=bin mode=0755
dir path=opt/mysoftware owner=root group=bin mode=0755
dir path=opt/mysoftware/bin owner=root group=bin mode=0755
file reloc/opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0755
dir path=opt/mysoftware/lib owner=root group=bin mode=0755
file reloc/opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir path=opt/mysoftware/man owner=root group=bin mode=0755
dir path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file reloc/opt/mysoftware/man/man1/mycmd.1 \
    path=opt/mysoftware/man/man1/mycmd.1 owner=root group=bin mode=0644
legacy pkg=MSFTmypkg arch=i386 category=system \
    desc="A sample SVR4 package of My Sample Package" \
    hotline="Please contact your local service provider" \
    name="My Sample Package" vendor="My Software Inc." \
    version=11.11,REV=2011.10.17.14.08
license install/copyright license=MSFTmypkg.copyright
```

Note the following points regarding the pkgsend generate output:

- The pkg.summary and pkg.description attributes were automatically created from data in the pkginfo file.
- A set action was generated from the extra parameter in the pkginfo file. This set action is set beneath the pkg.send.convert.* namespace. Use pkgmogrify(1) transforms to convert such attributes to more appropriate attribute names.
- A legacy action was generated from data in the pkginfo file.
- A license action was generated that points to the copyright file used in the SVR4 package.
- An error message was emitted regarding a scripting operation that cannot be converted.

The following check shows the error message and the non-zero return code from pkgsend generate:

```
$ pkgsend generate MSFTmypkg > /dev/null
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall
$ echo $?
1
```

The SVR4 package is using a postinstall script that cannot be converted directly to an IPS equivalent. The script must be manually inspected.

The postinstall script in the package has the following content:

```
#!/usr/bin/sh
catman -M /opt/mysoftware/man
```

You can achieve the same results as this script by using a restart_fmri actuator that points to an existing SMF service, svc:/application/man-index:default, as described in "Add Any Facets or Actuators That Are Needed" on page 39. See Chapter 7, "Automating System Change as Part of Package Installation," for a thorough discussion of actuators.

The pkgsend generate command also checks for the presence of class-action scripts and produces error messages that indicate which scripts should be examined.

In any conversion of a SVR4 package to an IPS package, the needed functionality probably can be implemented by using an existing action type or SMF service. See "Package Content: Actions" on page 20 for details about available action types. See Chapter 7, "Automating System Change as Part of Package Installation," for information about SMF and package actions.

Adding package metadata and resolving dependencies are done in the same way as described in "Creating and Publishing a Package" on page 34 and therefore are not discussed in this section. The next package creation step that might present unique issues for converted packages is the verification step.

Verify the Converted Package

A common source of errors when converting SVR4 packages is mismatched attributes between directories delivered in the SVR4 package and the same directories delivered by IPS packages.

In the SVR4 package in this example, the directory action for `/opt` in the sample manifest has different attributes than the attributes defined for this directory by the system packages.

The "Directory Actions" on page 23 section stated that all reference-counted actions must have the same attributes. When trying to install the version of `mypkg` that has been generated so far, the following error occurs:

```
# pkg install mypkg
Creating Plan /
pkg install: The requested change to the system attempts to install multiple actions
for dir 'opt' with conflicting attributes:

    1 package delivers 'dir group=bin mode=0755 owner=root path=opt':
        pkg://mypublisher/mypkg@1.0,5.11-0:20111017T020042Z
    3 packages deliver 'dir group=sys mode=0755 owner=root path=opt':
        pkg://solaris/developer/build/onbld@0.5.11,5.11-0.175.0.0.0.1.0:20111012T010101Z
        pkg://solaris/system/core-os@0.5.11,5.11-0.175.0.0.0.1.0:20111012T023456Z

These packages may not be installed together. Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

To catch the error before publishing the package, rather than at install time, use the `pkglint(1)` command with a reference repository, as shown in the following example:

```
$ pkglint -c ./cache -r file:///scratch/solaris-repo ./mypkg.mf.res
Lint engine setup...

PHASE                                            ITEMS
4                                            4292/4292
Starting lint run...

ERROR pkglint.dupaction007        path opt is reference-counted but has different attributes across 5
duplicates: group: bin -> mypkg group: sys -> developer/build/onbld system/core-os system/ldoms/ldomsmanager
```

Notice the error message about `path opt` having different attributes in different packages.

The extra `ldomsmanager` package that `pkglint` reports is in the reference package repository, but is not installed on the test system. The `ldomsmanager` package is not listed in the error reported previously by `pkg install` because that package is not installed.

Other Package Conversion Considerations

While it is possible to install SVR4 packages directly on an Oracle Solaris 11 system, you should create IPS packages instead. Installing SVR4 packages is an interim solution.

Apart from the `legacy` action described in "Legacy Actions" on page 29, no links exist between the two packaging systems, and SVR4 and IPS packages do not reference package metadata from each other.

IPS has commands such as `pkg verify` that can determine whether packaged content has been installed correctly. However, errors can result if another packaging system legitimately installs packages or runs install scripts that modify directories or files installed by IPS packages.

The IPS `pkg fix` and `pkg revert` commands can overwrite files delivered by SVR4 packages as well as by IPS packages, potentially causing the packaged applications to malfunction.

Commands such as `pkg install`, which normally check for duplicate actions and common attributes on reference-counted actions, might fail to detect potential errors when files from a different packaging system conflict.

With these potential errors in mind, and given the comprehensive package development tool chain in IPS, developing IPS packages instead of SVR4 packages is recommended for Oracle Solaris 11.

CHAPTER 3

Installing, Removing, and Updating Software Packages

This chapter describes how the IPS client works internally when installing, updating, and removing the software installed in an image.

Understanding how pkg(1) performs these operations is important for understanding the various errors that can occur and for more quickly resolving package dependency problems.

How Package Changes Are Performed

The following steps are executed when pkg is invoked to modify the software installed on the machine:

- Check the input for errors
- Determine the system end-state
- Run basic checks
- Run the solver
- Optimize the solver results
- Evaluate actions
- Download content
- Execute actions
- Process actuators

When executing these steps in the global zone, pkg can also operate on any non-global zones on the system. For example, pkg ensures that dependencies are correct between the global zone and non-global zones, and downloads content and executes actions as needed for non-global zones. Chapter 10, "Handling Non-Global Zones," discusses zones in detail.

Check Input for Errors

Basic error checking is performed on the options presented on the command line.

Determine the System End State

A description of the desired end state of the system is constructed. In the case of updating all packages in the image, the desired end state might be something like “all the packages currently installed, or newer versions of them.” In the case of package removal, the desired end state is “all the packages currently installed without this one.”

IPS attempts to determine what the user intends this end state to look like. In some cases, IPS might determine an end state that is not what the user intended, even though that end state does match what the user requested.

When troubleshooting, it is best to be as specific as possible. The following command is not specific:

```
# pkg update
```

If this command fails with a message such as “No updates available for this image,” then you might want to try a more specific command such as the following command:

```
# pkg update "*@latest"
```

This command defines the end state more precisely, and can produce more directed error messages.

Run Basic Checks

The desired end state of the system is reviewed to make sure that a solution is possible. During this basic review, `pkg` checks that a plausible version exists of all dependencies, and that desired packages do not exclude each other.

If an obvious error exists, then `pkg` prints an appropriate error message and exits.

Run the Solver

The solver forms the core of the computation engine used by `pkg(5)` to determine the packages that can be installed, updated, or removed, given the constraints in the image and constraints introduced by any new packages for installation.

This problem is an example of a *Boolean satisfiability problem*, and can be solved by a SAT solver.

The various possible choices for all the packages are assigned Boolean variables, and all the dependencies between those packages, any required packages, and so on, are cast as Boolean expressions in conjunctive normal form.

The set of expressions generated is passed to MiniSAT. If MiniSAT cannot find any solution, the error handling code attempts to walk the set of installed packages and the attempted operation and print the reasons that each possible choice was eliminated.

If the currently installed set of packages meets the requirements but no other set does, `pkg` reports that there is nothing to do.

As mentioned previously, the error message generation and specificity is determined by the inputs to `pkg`. Being as specific as possible in commands issued to `pkg` produces the most useful error messages.

If MiniSAT finds a possible solution, the optimization phase begins.

Optimize the Solver Results

The optimization phase is necessary because there is no way to describe some solutions as more desirable than others to a SAT solver. Instead, once a solution is found, IPS adds constraints to the problem to separate less desirable choices, and to separate the current solution as well. IPS then repeatedly invokes MiniSAT and repeats the above operation until no more solutions are found. The last successful solution is taken as the best one.

The difficulty of finding a solution is proportional to the number of possible solutions. Being more specific about the desired result produces solutions more quickly.

Once the set of package FMRIs that best satisfy the posed problem is found, the evaluation phase begins.

Evaluate Actions

In the evaluation phase, IPS compares the packages currently installed on the system with the end state, and compares package manifests of old and new packages to determine three lists:

- Actions that are being removed.
- Actions that are being added.
- Actions that are being updated.

The action lists are then updated in the following ways:

- Directory and link actions are reference counted, and mediated link processing is done.
- Hard links are marked for repair if their target file is updated. This is done because updating a target of a hard link in a manner that is safe for currently executing processes breaks the hard links.
- Editable files moving between packages are correctly handled so that any user edits are not lost.

- Action lists are sorted so that removals, additions, and updates occur in the correct order.

All currently installed packages are then cross-checked to make sure that no packages conflict. Example conflicts include two packages that deliver a file to the same location, or two packages that deliver the same directory with different directory attributes.

If conflicts exist, the conflicts are reported and `pkg` exits with an error message.

Finally, the action lists are scanned to determine whether any SMF services need to be restarted if this operation is performed, whether this change can be applied to a running system, whether the boot archive needs to be rebuilt, and whether the amount of space required is available.

Download Content

If `pkg` is running without the `-n` flag, processing continues to the download phase.

For each action that requires content, IPS downloads any required files by hash and caches them. This step can take some time if the amount of content to be retrieved is large.

Once downloading is complete, if the change is to be applied to a live system (the image is rooted at `/`), and a reboot is required, the running system is cloned and the target image is switched to the clone.

Execute Actions

Executing actions involves actually performing the install or remove methods specific to each action type on the image.

Execution begins with all the removal actions being executed. If any unexpected content is found in directories being removed from the system, that content is placed in `/var/pkg/lost+found`.

Execution then proceeds to install and update actions. Note that all the actions have been blended across all packages. Thus all the changes in a single package operation are applied to the system at once rather than package by package. This permits packages to depend on each other and exchange content safely. For details on how files are updated, see "File Actions" on page 21.

Process Actuators

If the changes are being applied to a live system, any pending actuators are executed at this point. These are typically SMF service restarts and refreshes. Once these are launched, IPS updates the local search indices. Actuators are discussed in detail in Chapter 7, "Automating System Change as Part of Package Installation."

Update Boot Archive

If necessary, the boot archive is updated.

CHAPTER 4

Specifying Package Dependencies

Dependencies define how packages are related. This chapter explains the different types of IPS dependencies and how they can be used to construct working software systems.

IPS provides a variety of different dependency types as discussed in "Depend Actions" on page 27. This chapter provides more detail about how each dependency type can be used to control the software that is installed.

Dependency Types

In IPS, a package cannot be installed unless all package dependencies are satisfied. IPS allows packages to be mutually dependent (to have circular dependencies). IPS also allows packages to have different kinds of dependencies on the same package at the same time.

Each section in this chapter contains an example `depend` action as it would appear in a manifest during package creation.

require Dependency

The most basic type of dependency is the `require` dependency. These dependencies are typically used to express functional dependencies such as libraries, or interpreters such as Python or Perl.

If a package `A@1.0` contains a `require` dependency on package `B@2`, then if `A@1.0` is installed, the `B` package at version 2 or higher must also be installed. This acceptance of higher versioned packages reflects the implicit expectation of binary compatibility in newer versions of existing packages.

If any version of the package named in the `depend` action is acceptable, you can omit the version portion of the specified FMRI.

An example `require` dependency is:

```
depend fmri=pkg:/system/library type=require
```

require-any Dependency

The `require-any` dependency is used if more than one package can satisfy a functional requirement. IPS chooses one of the packages to install if the dependency is not already satisfied.

For example, you could use a `require-any` dependency to ensure that at least one version of Perl is installed on the system. The versioning is handled in the same manner as for the `require` dependency.

An example `require-any` dependency is:

```
depend type=require-any fmri=pkg:/editor/gnu-emacs/gnu-emacs-gtk \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-no-x11 \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-x11
```

optional Dependency

The `optional` dependency specifies that if the given package is installed, it must be at the given version or greater.

This type of dependency is typically used to handle cases where packages transfer content. In this case, each version of the package post-transfer would contain an `optional` dependency on the post-transfer version of the other package, so that it would be impossible to install incompatible versions of the two packages. Omitting the version on an `optional` dependency makes the dependency meaningless, but is permitted.

An example `optional` dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.9.99 type=optional
```

conditional Dependency

The `conditional` dependency has a `predicate` attribute and an `fmri` attribute. If the package specified in the value of the `predicate` attribute is present on the system at the specified or greater version, the `conditional` dependency is treated as a `require` dependency on the package in the `fmri` attribute. If the package specified in the `predicate` attribute is not present on the system or is present at a lower version, the `conditional` dependency is ignored.

The `conditional` dependency is most often used to install optional extensions to a package if the requisite base packages are present on the system.

For example, an editor package that has both X11 and terminal versions might place the X11 version in a separate package, and include a `conditional` dependency on the X11 version from the text version with the existence of the requisite X client library package as the `predicate`.

An example `conditional` dependency is:

```
depend type=conditional fmri=library/python-2/pycurl-26 \
    predicate=runtime/python-26
```

group Dependency

The `group` dependency is used to construct groups of packages.

The `group` dependency ignores the version specified. Any version of the named package satisfies this dependency.

The named package is required unless the package has been the object of one of the following actions:

- The package has been placed on the avoid list. See the `pkg(1)` man page for information about the avoid list.
- The package has been rejected with `pkg install --reject`.
- The package has been uninstalled with `pkg uninstall`.

These three options enable administrators to deselect packages that are the subject of a `group` dependency. If any of these three options has been used, IPS will not reinstall the package during an update unless the package was subsequently required by another dependency. If the new dependency is removed by another subsequent operation, then the package is uninstalled again.

A good example of how to use these dependencies is to construct packages containing `group` dependencies on packages that are needed for typical uses of a system. Some examples might be `solaris-large-server`, `solaris-desktop`, or `developer-gnu`. "Oracle Solaris Group Packages" on page 114 shows a set of Oracle Solaris packages that deliver `group` dependencies.

Installing group packages provides confidence that over subsequent updates to newer versions of the OS, the appropriate packages will be added to the system.

An example `group` dependency is:

```
depend fmri=package/pkg type=group
```

origin Dependency

The `origin` dependency exists to resolve upgrade issues that require intermediate transitions. The default behavior is to specify the minimum version of a package (if installed) that must be present on the system being updated.

For example, a typical use might be a database package version 5 that supports upgrade from version 3 or greater, but not earlier versions. In this case, version 5 would have an `origin` dependency on itself at version 3. Thus, if version 5 was being freshly installed, installation would proceed. However, if version 1 of the package was installed, the package could not be upgraded directly to version 5. In this case, `pkg update database-package` would not select version 5 but instead would select version 3 as the latest possible version to which to upgrade.

The behavior of the `origin` dependency can be modified by setting the `root-image` attribute to `true`. In this case, the named package must be at the specified version or greater if it is present in the running system, rather than the image being updated. This is generally used for operating system issues such as dependencies on boot block installers.

An example `origin` dependency is:

```
depend fmri=pkg:/database/mydb@3.0 type=origin
```

incorporate Dependency

The `incorporate` dependency specifies that if the given package is installed, it must be at the given version, to the given version accuracy. For example, if the dependent FMRI has a version of 1.4.3, then no version less than 1.4.3 or greater than or equal to 1.4.4 satisfies the dependency. Version 1.4.3.7 does satisfy this example dependency.

The common way to use `incorporate` dependencies is to put many of them in the same package to define a surface in the package version space that is compatible. Packages that contain such sets of `incorporate` dependencies are often called incorporations. Incorporations are typically used to define sets of software packages that are built together and are not separately versioned. The `incorporate` dependency is heavily used in Oracle Solaris to ensure that compatible versions of software are installed together.

An example `incorporate` dependency is:

```
depend type=incorporate \
    fmri=pkg:/driver/network/ethernet/e1000g@0.5.11,5.11-0.175.0.0.0.2.1
```

parent Dependency

The parent dependency is used for zones or other child images. In this case, the dependency is only checked in the child image, and specifies a package and version that must be present in the parent image or global zone. The version specified must match to the level of precision specified.

For example, if the parent dependency is on A@2.1, then any version of A beginning with 2.1 matches. This dependency is often used to require that packages are kept in sync between non-global zones and the global zone. As a shortcut, the special package name feature/package/dependency/self is used as a synonym for the exact version of the package that contains this dependency.

The parent dependency is used to keep key operating system components, such as libc.so.1, installed in the non-global zone synchronized with the kernel installed in the global zone. The parent dependency is also discussed in Chapter 10, “Handling Non-Global Zones.”

An example parent dependency is:

```
depend type=parent fmri=feature/package/dependency/self \
    variant.opensolaris.zone=nonglobal
```

exclude Dependency

The package that contains the exclude dependency cannot be installed if the dependent package is installed in the image at the specified version level or greater.

If the version is omitted from the FMRI of an exclude dependency, then no version of the excluded package can be installed concurrently with the package specifying the dependency.

The exclude dependency is seldom used. These constraints can be frustrating to administrators, and should be avoided where possible.

An example exclude dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.10.99 type=exclude
```

Constraints and Freezing

Through the careful use of the various types of depend actions described above, packages can define the ways in which they are allowed to be upgraded.

Constraining Installable Package Versions

Typically, you want a set of packages installed on a system to be supported and upgraded together: Either all packages in the set are updated, or none of the packages in the set are updated. This is the reason for using the `incorporate` dependency.

The following three partial package manifests show the relationship between the `foo` and `bar` packages and the `myincorp` incorporation package.

The following excerpt is from the `foo` package manifest:

```
set name=pkg.fmri value=foo@1.0
dir path=foo owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

The following excerpt is from the `bar` package manifest:

```
set name=pkg.fmri value=bar@1.0
dir path=bar owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

The following excerpt is from the `myincorp` package manifest:

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=foo@1.0 type=incorporate
depend fmri=bar@1.0 type=incorporate
```

The `foo` and `bar` packages both have a `require` dependency on the `myincorp` incorporation. The `myincorp` package has `incorporate` dependencies that constrain the `foo` and `bar` packages in the following ways:

- The `foo` and `bar` packages can be upgraded to at most version 1.0: to the level of granularity defined by the version number specified in the dependency.
- If the `foo` and `bar` packages are installed, they must be at least at version 1.0 or greater.

The `incorporate` dependency on version 1.0 allows version 1.0.1 or 1.0.2.1, for example, but does not allow version 1.1, 2.0, or 0.9, for example. When an updated incorporation package is installed that specifies `incorporate` dependencies at a higher version, the `foo` and `bar` packages are allowed to update to those higher versions.

Because `foo` and `bar` both have `require` dependencies on the `myincorp` package, the incorporation package must always be installed.

Relaxing Constraints on Installable Package Versions

In some situations, you might want to relax an incorporation constraint.

Perhaps `bar` can function independently of `foo`, but you want `foo` to remain within the series of versions defined by the `incorporate` dependency in the incorporation.

You can use facets to relax incorporation constraints, allowing the administrator to effectively disable certain `incorporate` dependencies. Facets are discussed in more detail in Chapter 5, "Allowing Variations." Briefly, facets are special attributes that can be applied to actions within a package to enable authors to mark those actions as optional.

When actions are marked with facet attributes in this manner, the actions that contain those facets can be enabled or disabled using the `pkg change-facet` command.

By convention, facets that optionally install `incorporate` dependencies are named `facet.version-lock.`*name*, where *name* is the name of the package that contains that `depend` action.

Using the example above, the `myincorp` package manifest could contain the following lines:

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=foo@1.0 type=incorporate
depend fmri=bar@1.0 type=incorporate facet.version-lock.bar=true
```

By default, this incorporation includes the `depend` action on the `bar` package, constraining `bar` to version 1.0. The following command relaxes this constraint:

```
# pkg change-facet version-lock.bar=false
```

After successful execution of this command, the `bar` package is free from the incorporation constraints and can be upgraded to version 2.0 if necessary.

Freezing Installable Package Versions

So far, the discussion has been around constraints applied during the package authoring process by modifying the package manifests. The administrator can also apply constraints to the system at runtime.

Using the `pkg freeze` command, the administrator can prevent a given package from being changed from either its current installed version, including time stamp, or a version specified on the command line. This capability is effectively the same as an `incorporate` dependency.

See the `pkg(1)` man page for more information about the `freeze` command.

To apply more complex dependencies to an image, create and install a package that includes those dependencies.

◆ ◆ ◆ CHAPTER 5

Allowing Variations

This chapter explains how to provide different installation options to the end user.

Mutually Exclusive Software Components

Oracle Solaris supports multiple architectures, and one common error made with the SVR4 packaging system was the accidental installation of packages for an incorrect architecture. Maintaining separate IPS software repositories for each supported architecture is unappealing to ISVs and error prone for software users. As a result, IPS supports installation of a single package on multiple architectures.

The mechanism that implements this feature is called a *variant*. A variant enables the properties of the target image to determine which software components are actually installed.

A variant has two parts: its name, and the list of possible values. The variants defined in Oracle Solaris 11 are shown in the following table.

Variant Name	Possible Values
`variant.arch`	`sparc`, `i386`
`variant.opensolaris.zone`	`global`, `nonglobal`
`variant.debug.*`	`true`, `false`

Variants appear in the following two places in a package:

- A `set` action names the variant and defines the values that apply to this package.
- Any action that can only be installed for a subset of the variant values named in the `set` action has a tag that specifies the name of the variant and the value on which this action is installed.

For example, a package that delivers the symbolic link /var/ld/64 might include the following definitions:

```
set name=variant.arch value=sparc value=i386
dir group=bin mode=0755 owner=root path=var/ld
dir group=bin mode=0755 owner=root path=var/ld/amd64 \
    variant.arch=i386
dir group=bin mode=0755 owner=root path=var/ld/sparcv9 \
    variant.arch=sparc
link path=var/ld/32 target=.
link path=var/ld/64 target=sparcv9 variant.arch=sparc
link path=var/ld/64 target=amd64 variant.arch=i386
```

Note that components that are delivered on both SPARC and x86 receive no variant tag, but components delivered to one architecture or the other receive the appropriate tag. Actions can contain multiple tags for different variant names. For example, a package might include both debug and nondebug binaries for both SPARC and x86.

In Oracle Solaris, kernel components are commonly elided from packages installed in zones, since kernel components serve no useful purpose in a non-global zone. Thus, the kernel components are marked with the opensolaris.zone variant set to global so that they are not installed in non-global zones. This is typically done in the manifest during publication by using a pkgmogrify(1) rule. The packages from the i386 and sparc builds are marked for zones, and then pkgmerge(1) merges packages from the sparc and i386 builds. This is far more reliable and faster than attempting to construct such packages manually.

Package developers cannot define new variants. However, developers can provide debug versions of components, tagged with a variant.debug.* variant, and users can select that variant if problems arise. The variant.debug.* portion of the variant namespace is predefined to have a default value of false. Remember that variants are set per image, so be sure to select a suitable name that is unique at the appropriate resolution for that piece of software.

Variant tags are applied to any actions that differ between architectures during merging, including dependencies and set actions. Packages that are marked as not supporting one of the variant values of the current image are not considered for installation.

The pkgmerge(1) man page provides several examples of merging packages. The pkgmerge command merges across multiple different variants at the same time if needed.

Optional Software Components

Some portions of your software that belong with the main body might be optional, and some users might not want to install them. Examples include localization files for different locales, man pages and other documentation, and header files needed only by developers or DTrace users.

Traditionally, optional content has been delivered in separate packages with identifiers such as -dev or -devel appended to the package name. Administrators installed optional content by installing these optional packages. One problem with this solution is that the administrator must discover optional packages to install by examining lists of available packages.

IPS implements a mechanism called *facets* to deliver optional package content. Facets are similar to variants: Each facet has a name and a value, and actions can contain multiple tags for different facet names. In the image, the default value for all facets is true, and the value of a particular facet can be explicitly set to either true or false. The facet namespace is hierarchical. The `pkg` client implicitly sets `facet.*` to true for the image. The value of a particular facet in the image is the value of the longest matching facet name.

The following example shows how the administrator can include man pages but exclude all other documentation from being installed in this image. Man pages and other documentation can be in the same package with software and other content that the administrator wants to install. In the package manifests, man pages are tagged with `facet.doc.man=true`. Other documentation actions might be tagged with `facet.doc.pdf=true` or `facet.doc.html=true`, for example. In the image, the administrator can use the following commands to include the man pages but exclude all other documentation:

```
# pkg change-facet facet.doc.*=false
# pkg change-facet facet.doc.man=true
```

Similarly, actions in package manifests can be tagged with locale facets such as `facet.locale.de=true` or `facet.locale.fr=true`. The following commands install only the German localization in this image:

```
# pkg change-facet facet.locale.*=false
# pkg change-facet facet.locale.de=true
```

If an action contains multiple facet tags, the action is installed if the value of any of the facet tags is true. Use the `pkg facet` command to display the facets that have been explicitly set in the image.

```
$ pkg facet
FACETS          VALUE
facet.doc.*     False
facet.doc.man   True
facet.locale.*  False
facet.locale.de True
```

Use `pkgmogrify` to quickly and accurately add facet tags to your package manifests, using regular expressions to match the different types of files. This is described in detail in Chapter 6, "Modifying Package Manifests Programmatically."

Facets can also be used to manage dependencies, turning dependencies on and off depending on whether the facet is set. See "Constraints and Freezing" on page 61 for a discussion of `facet.version-lock.*`.

The following facets might be useful for software developers:

```
facet.devel             facet.locale.es_BO      facet.locale.lt_LT
facet.doc               facet.locale.es_CL      facet.locale.lv
facet.doc.man           facet.locale.es_CO      facet.locale.lv_LV
facet.doc.pdf           facet.locale.es_CR      facet.locale.mk
facet.doc.info          facet.locale.es_DO      facet.locale.mk_MK
facet.doc.html          facet.locale.es_EC      facet.locale.ml
facet.locale.*          facet.locale.es_ES      facet.locale.ml_IN
facet.locale.af         facet.locale.es_GT      facet.locale.mr
facet.locale.af_ZA      facet.locale.es_HN      facet.locale.mr_IN
facet.locale.ar         facet.locale.es_MX      facet.locale.ms
facet.locale.ar_AE      facet.locale.es_NI      facet.locale.ms_MY
facet.locale.ar_BH      facet.locale.es_PA      facet.locale.mt
facet.locale.ar_DZ      facet.locale.es_PE      facet.locale.mt_MT
facet.locale.ar_EG      facet.locale.es_PR      facet.locale.nb
facet.locale.ar_IQ      facet.locale.es_PY      facet.locale.nb_NO
facet.locale.ar_JO      facet.locale.es_SV      facet.locale.nl
facet.locale.ar_KW      facet.locale.es_US      facet.locale.nl_BE
facet.locale.ar_LY      facet.locale.es_UY      facet.locale.nl_NL
facet.locale.ar_MA      facet.locale.es_VE      facet.locale.nn
facet.locale.ar_OM      facet.locale.et         facet.locale.nn_NO
facet.locale.ar_QA      facet.locale.et_EE      facet.locale.no
facet.locale.ar_SA      facet.locale.eu         facet.locale.or
facet.locale.ar_TN      facet.locale.fi         facet.locale.or_IN
facet.locale.ar_YE      facet.locale.fi_FI      facet.locale.pa
facet.locale.as         facet.locale.fr         facet.locale.pa_IN
facet.locale.as_IN      facet.locale.fr_BE      facet.locale.pl
facet.locale.az         facet.locale.fr_CA      facet.locale.pl_PL
facet.locale.az_AZ      facet.locale.fr_CH      facet.locale.pt
facet.locale.be         facet.locale.fr_FR      facet.locale.pt_BR
facet.locale.be_BY      facet.locale.fr_LU      facet.locale.pt_PT
facet.locale.bg         facet.locale.ga         facet.locale.ro
facet.locale.bg_BG      facet.locale.gl         facet.locale.ro_RO
facet.locale.bn         facet.locale.gu         facet.locale.ru
facet.locale.bn_IN      facet.locale.gu_IN      facet.locale.ru_RU
facet.locale.bs         facet.locale.he         facet.locale.ru_UA
facet.locale.bs_BA      facet.locale.he_IL      facet.locale.rw
facet.locale.ca         facet.locale.hi         facet.locale.sa
facet.locale.ca_ES      facet.locale.hi_IN      facet.locale.sa_IN
facet.locale.cs         facet.locale.hr         facet.locale.sk
facet.locale.cs_CZ      facet.locale.hr_HR      facet.locale.sk_SK
facet.locale.da         facet.locale.hu         facet.locale.sl
facet.locale.da_DK      facet.locale.hu_HU      facet.locale.sl_SI
facet.locale.de         facet.locale.hy         facet.locale.sq
facet.locale.de_AT      facet.locale.hy_AM      facet.locale.sq_AL
facet.locale.de_BE      facet.locale.id         facet.locale.sr
facet.locale.de_CH      facet.locale.id_ID      facet.locale.sr_ME
facet.locale.de_DE      facet.locale.is         facet.locale.sr_RS
facet.locale.de_LI      facet.locale.is_IS      facet.locale.sv
facet.locale.de_LU      facet.locale.it         facet.locale.sv_SE
facet.locale.el         facet.locale.it_CH      facet.locale.ta
facet.locale.el_CY      facet.locale.it_IT      facet.locale.ta_IN
facet.locale.el_GR      facet.locale.ja         facet.locale.te
facet.locale.en         facet.locale.ja_JP      facet.locale.te_IN
facet.locale.en_AU      facet.locale.ka         facet.locale.th
facet.locale.en_BW      facet.locale.ka_GE      facet.locale.th_TH
facet.locale.en_CA      facet.locale.kk         facet.locale.tr
facet.locale.en_GB      facet.locale.kk_KZ      facet.locale.tr_TR
facet.locale.en_HK      facet.locale.kn         facet.locale.uk
facet.locale.en_IE      facet.locale.kn_IN      facet.locale.uk_UA
facet.locale.en_IN      facet.locale.ko         facet.locale.vi
facet.locale.en_MT      facet.locale.ko_KR      facet.locale.vi_VN
facet.locale.en_NZ      facet.locale.ks         facet.locale.zh
facet.locale.en_PH      facet.locale.ks_IN      facet.locale.zh_CN
facet.locale.en_SG      facet.locale.ku         facet.locale.zh_HK
facet.locale.en_US      facet.locale.ku_TR      facet.locale.zh_SG
facet.locale.en_ZW      facet.locale.ky         facet.locale.zh_TW
facet.locale.eo         facet.locale.ky_KG
facet.locale.es_AR      facet.locale.lg
```

CHAPTER 6

Modifying Package Manifests Programmatically

This chapter explains how package manifests can be machine edited to automatically annotate and check the manifests.

Chapter 2, "Packaging Software With IPS," covers all the techniques that are necessary to publish a package. This chapter provides additional information that can help you publish a large package, publish a large number of packages, or republish packages over a period of time.

Your package might contain many actions that need to be tagged with variants or facets as discussed in Chapter 5, "Allowing Variations," or that need to be tagged with service restarts as discussed in Chapter 7, "Automating System Change as Part of Package Installation." Rather than edit package manifests manually or write a script or program to do this work, use the IPS `pkgmogrify` utility to transform the package manifests quickly, accurately, and repeatably.

The `pkgmogrify` utility applies two types of rules: transform and include. Transform rules modify actions. Include rules cause other files to be processed. The `pkgmogrify` utility reads these rules from a file and applies them to the specified package manifest.

Transform Rules

This section shows an example transform rule and describes the parts of all transform rules.

In Oracle Solaris, files delivering in a subdirectory named `kernel` are treated as kernel modules and are tagged as requiring a reboot. The following tag is applied to actions whose `path` attribute value includes `kernel`:

```
reboot-needed=true
```

To apply this tag, the following rule is specified in the `pkgmogrify` rule file:

```
<transform file path=.*kernel/.+ -> default reboot-needed true>
```

delimiters	The rule is enclosed with < and >. The portion of the rule to the left of the `->` is the selection section or matching section. The portion to the right of the `->` is the execution section of the operation.
`transform`	The type of the rule.
`file`	Apply this rule only to `file` actions. This is called the selection section of the rule.
`path=.*kernel/.+`	Transform only `file` actions with a `path` attribute that matches the regular expression `path=.*kernel/.+`. This is called the matching section of the rule.
`default`	Add the attribute and value that follow `default` to any matching action that does not already have a value set for that attribute.
`reboot-needed`	The attribute being set.
`true`	The value of the attribute being set.

The selection or matching section of a transform rule can restrict by action type and by action attribute value. See the `pkgmogrify` man page for detail about how these matching rules work. Typical uses are for selecting actions that deliver to specified areas of the file system. For example, in the following rule, *operation* could be used to ensure that `usr/bin` and everything delivered inside `usr/bin` defaults to the correct user or group.

```
<transform file dir link hardlink path=usr/bin.* -> operation>
```

The `pkgmogrify(1)` man page describes the many operations that `pkgmogrify` can perform to add, remove, set, and edit action attributes as well as add and remove entire actions.

Include Rules

Include rules enable transforms to be spread across multiple files and subsets reused by different manifests. Suppose you need to deliver two packages: `A` and `B`. Both packages should have their `source-url` set to the same URL, but only package `B` should have its files in `/etc` set to be `group=sys`.

The manifest for package `A` should specify an include rule that pulls in the file with the `source-url` transform. The manifest for package `B` should specify an include rule that pulls in the file containing the file group setting transform. Finally, an include rule that pulls in the file with the `source-url` transform should be added either to either package `B` or to the file with the transform that sets the group.

Transform Order

Transforms are applied in the order in which they are encountered in a file. The ordering can be used to simplify the matching portions of transforms.

Suppose all files delivered in `/foo` should have a default group of `sys`, except those files delivered in `/foo/bar`, which should have a default group of `bin`.

You could write a complex regular expression that matches all paths that begin with `/foo` except for paths that begin with `/foo/bar`. Using the ordering of transforms makes this matching much simpler.

When ordering default transforms, always go from most specific to most general. Otherwise the latter rules will never be used.

For this example, use the following two rules:

```
<transform file path=foo/bar/.* -> default group bin>
<transform file path=foo/.* -> default group sys>
```

Using transforms to add an action using the matching described above would be difficult since you would need to find a pattern that matched each package delivered once and only once. The `pkgmogrify` tool creates synthetic actions to help with this issue. As `pkgmogrify` processes manifests, for each manifest that sets the `pkg.fmri` attribute, a synthetic `pkg` action is created by `pkgmogrify`. You can match against the `pkg` action as if it were actually in the manifest.

For example, suppose you wanted to add to every package an action containing the web site `example.com`, where the source code for the delivered software can be found. The following transform accomplishes that:

```
<transform pkg -> emit set info.source-url=http://example.com>
```

Packaged Transforms

As a convenience to developers, a set of the transforms that were used when packaging the Oracle Solaris OS are available in the following files in `/usr/share/pkg/transforms`:

`developer`	Sets `facet.devel` on `*.h` header files delivered to `/usr/.*/include`, archive and lint libraries, `pkg-config`(1) data files, and `autoconf`(1) macros.
`documentation`	Sets a variety of `facet.doc.*` facets on documentation files.
`locale`	Sets a variety of `facet.locale.*` facets on files that are locale-specific.

`smf-manifests`	Adds a `restart_fmri` actuator that points to the `svc:/system/manifest-import:default` on any packaged SMF manifests so that the system will import that manifest after the package is installed.

◆ ◆ ◆ CHAPTER 7

Automating System Change as Part of Package Installation

This chapter explains how to use the Service Management Facility (SMF) to automatically handle any necessary system changes that should occur as a result of package installation.

Specifying System Changes on Package Actions

First determine which actions should cause a change to the system when they are installed, updated, or removed. For example, some system changes are needed to implement the software self-assembly concept described in "Software Self-Assembly" on page 13.

For each of those package actions, determine which existing SMF service provides the necessary system change. Alternatively, write a new service that provides the needed functionality and ensure that service is delivered to the system as described in "Delivering an SMF Service" on page 76.

When you have determined the set of actions that should cause a change to the system when they are installed, tag those actions in the package manifest to cause that system change to occur. The value of a tag that causes system change to occur is called an *actuator*.

The following actuator tags can be added to any action in a manifest:

`reboot-needed`	This actuator takes the value `true` or `false`. This actuator declares that update or removal of the tagged action must be performed in a new boot environment if the package system is operating on a live image. Creation of a new boot environment is controlled by the `be-policy` image property. See the "Image Properties" section in the `pkg(1)` man page for more information about the `be-policy` property.
SMF Actuators	These actuators are related to SMF services. SMF actuators take a single service FMRI as a value, possibly including globbing characters to match multiple FMRIs. If the same service FMRI is

tagged by multiple actions, possibly across multiple packages being operated on, IPS only triggers that actuator once.

The following list of SMF actuators describes the effect on the service FMRI that is the value of each named actuator:

`disable_fmri`	Disable the specified service prior to performing the package operation.
`refresh_fmri`	Refresh the specified service after completing the package operation.
`restart_fmri`	Restart the specified service after completing the package operation.
`suspend_fmri`	Temporarily suspend the specified service prior to performing the package operation, and enable the service after completing the package operation.

Delivering an SMF Service

To deliver a new SMF service, create a package that delivers the SMF manifest file and method script.

This section first discusses the general case of delivering any new SMF service, and then discusses the specific case of delivering a service that runs once. Finally, this section presents some tips for self-assembly of these service packages.

Delivering a New SMF Service

A package that delivers a new SMF service usually needs a system change.

In SVR4 packaging, post-install scripting ran an SMF command to restart the `svc:/system/manifest-import:default` service.

In IPS, the action that delivers the manifest file into `lib/svc/manifest` or `var/svc/manifest` should be tagged with the following actuator:

```
restart_fmri=svc:/system/manifest-import:default
```

This actuator ensures that when the manifest is added, updated, or removed, the `manifest-import` service is restarted, causing the service delivered by that SMF manifest to be added, updated, or removed.

If the package is added to a live system, this action is performed once all packages have been added to the system during that packaging operation. If the package is added to an alternate boot environment, this action is performed during the first boot of that boot environment.

Delivering a Service that Runs Once

A package that needs to perform a one-time configuration of the new software environment should deliver an SMF service to perform that configuration.

The package that delivers the application should include the following actions:

```
file path=opt/myapplication/bin/run-once.sh owner=root group=sys mode=0755
file path=lib/svc/manifest/application/myapplication-run-once.xml owner=root group=sys \
mode=0644 restart_fmri=svc:/system/manifest-import:default
```

The SMF method script for the service can contain anything that is needed to further configure the application or modify the system so that the application runs efficiently. In this example, the method script writes a simple log message:

```
#!/usr/bin/sh
. /lib/svc/share/smf_include.sh
assembled=$(/usr/bin/svcprop -p config/assembled $SMF_FMRI)
if [ "$assembled" == "true" ] ; then
    exit $SMF_EXIT_OK
fi
svccfg -s $SMF_FMRI setprop config/assembled = true
svccfg -s $SMF_FMRI refresh
echo "This is output from our run-once method script"
```

Generally, the SMF service should only perform work if the application has not already been configured. This example method script checks `config/assembled`. An alternative approach is to package the service separately from the application, and then use the method script to remove the package that contains the service.

When testing a method script, run `pkg verify` before and after installing the package that runs the actuator. Compare the output of each run to ensure that the script does not attempt to modify any files that are not marked as editable.

The following shows the SMF service manifest for this example:

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">
<service_bundle type='manifest' name='MyApplication:run-once'>
<service
    name='application/myapplication/run-once'
    type='service'
    version='1'>
    <single_instance />
    <dependency
```

```
        name='fs-local'
        grouping='require_all'
        restart_on='none'
        type='service'>
            <service_fmri value='svc:/system/filesystem/local:default' />
    </dependency>
    <dependent
        name='myapplication_self-assembly-complete'
        grouping='optional_all'
        restart_on='none'>
        <service_fmri value='svc:/milestone/self-assembly-complete' />
    </dependent>
    <instance enabled='true' name='default'>
        <exec_method
            type='method'
            name='start'
            exec='/opt/myapplication/bin/run-once.sh'
            timeout_seconds='0'/>
        <exec_method
            type='method'
            name='stop'
            exec=':true'
            timeout_seconds='0'/>
        <property_group name='startd' type='framework'>
            <propval name='duration' type='astring' value='transient' />
        </property_group>
        <property_group name='config' type='application'>
            <propval name='assembled' type='boolean' value='false' />
        </property_group>
    </instance>
</service>
</service_bundle>
```

Note that the SMF service has a `startd/duration` property set to `transient` so that `svc.startd`(1M) does not track processes for this service. Also note that the service adds itself as a dependency to the `self-assembly-complete` system milestone.

Supporting Package Self-Assembly in SMF Methods

This section provides some additional tips to support package self-assembly when writing SMF methods.

Testing Whether a Configuration File Recompile Is Necessary

If compiling a configuration file from packaged configuration file fragments is expensive to perform each time the method script runs, consider using the following test in the method script.

Run `ls -t` on a directory of packaged configuration file fragments, and then use `head -1` to select the most recently changed version. Compare the time stamp of this file with the time stamp of the unpackaged configuration file that is compiled from those fragments to determine whether the service needs to recompile the configuration file.

Limiting the Time To Wait for Self-Assembly To Complete

The example SMF service manifest shown above defines `timeout_seconds='0'` for the `start` method. This means that SMF will wait indefinitely for self-assembly to complete.

To assist in debugging, you might want to impose a finite timeout on self-assembly processes, enabling SMF to drop the service to maintenance if something goes wrong.

◆ ◆ ◆ CHAPTER 8

Advanced Topics For Package Updating

This chapter discusses renaming, merging, and splitting packages, moving package contents, delivering multiple implementations of an application, and sharing information across boot environments.

Avoiding Conflicting Package Content

For performance reasons, the solver works purely on the dependency information specified in packages. For most update operations, this information is sufficient to enable IPS to automatically install correct updated packages.

Packages whose dependencies indicate that they can be installed at the same time but whose content conflicts, cause conflict checking to fail in pre-installation. If conflict checking fails, the end user must try to fix the problem, perhaps by manually specifying different versions of some packages. An example of conflicting content is two packages that install the same file.

The package developer must ensure that conflicting packages cannot be installed due to constraining dependencies. Use the `pkglint` utility to help discover such conflicts. See "Verify the Package" on page 41 and the `pkglint(1)` man page for more information about `pkglint`.

Renaming, Merging and Splitting Packages

The desired organization of a software component can change because of mistakes in the original packages, changes in the product or its usage over time, or changes in the surrounding software environment. Sometimes just the name of a package needs to change. When planning such changes, consider the user who is performing an upgrade, to ensure that unintended side effects do not occur.

Three types of package reorganization are discussed in this section, in order of increasingly complex considerations for `pkg update`:

1. Renaming single packages
2. Merging two packages
3. Splitting a package

Renaming a Single Package

Renaming a single package is straightforward. IPS provides a mechanism to indicate that a package has been renamed.

To rename a package, publish a new version of the existing package with the following two actions:

- A `set` action in the following form:

  ```
  set name=pkg.renamed value=true
  ```
- A `require` dependency on the new package.

A renamed package cannot deliver content other than `depend` or `set` actions.

The new package must ensure that it cannot be installed at the same time as the original package before the rename. If both packages are covered by the same incorporation dependency, this restriction is automatic. If not, the new package must contain an `optional` dependency on the old package at the renamed version. This ensures that the solver will not select both packages, which would fail conflict checking.

A user who installs this renamed package automatically receives the new named package, since it is a dependency of the old version. If a renamed package is not depended upon by any other packages, it is automatically removed from the system. The presence of older software can cause a number of renamed packages to be shown as installed. When that older software is removed, the renamed packages are automatically removed as well.

Packages can be renamed multiple times without issue, though this is not recommended since it can be confusing to users.

Merging Two Packages

Merging packages is straightforward as well. The following two cases are examples of merging packages:

- One package absorbs another package at the renamed version.
- Two packages are renamed to the same new package name.

One Package Absorbs Another

Suppose package A@2 must absorb package B@3. To do this, rename package B to package A@2. Remember to include an optional dependency in A@2 on B@3, unless both packages are incorporated so that they update together as described above. A user upgrading B to B@3 now gets A installed since A has absorbed B.

Two Packages Are Renamed

In this case, rename both packages to the name of the new merged package, including two optional dependencies on the old packages in the new one if they are not otherwise constrained.

Splitting a Package

When you split a package, rename each resulting new package as described in "Renaming a Single Package" on page 82. If one of the resulting new packages is not renamed, the pre-split and post-split versions of that package are not compatible and might violate dependency logic when the end user tries to update the package.

Rename the original package, and include require dependencies on all new packages that resulted from the split. This ensures that any package that had a dependency on the original package will get all the new pieces.

Some components of the split package can be absorbed into existing packages as a merge. See "One Package Absorbs Another" on page 83.

Obsoleting Packages

Package obsoletion is the mechanism by which packages are emptied of contents and removed from the system. An obsoleted package does not satisfy require dependencies. Update fails if an installed package has a require dependency on a package that has become obsolete, unless a newer version of the installed package is available that does not contain the require dependency.

A package is made obsolete by publishing a new version with no content except for the following set action:

```
set name=pkg.obsolete value=true
```

A package can be made non-obsolete by publishing newer versions. Users who update when an installed package is obsolete lose that package. Users who updated before the package was obsolete and do not update again until after a newer version of the package is published are updated to that newer version.

Preserving Editable Files that Migrate

One common issue with updating packages is the migration of editable files, either moving between packages or changing location in the installed file system.

Migrating editable files between packages.
: IPS attempts to migrate editable files that move between packages if the file name and file path have not changed. Renaming a package is an example of moving files between packages.

Migrating editable files in the file system.
: If the file path changes, ensure the `original_name` attribute is assigned to preserve the user's customizations of the file.

  If the `file` action in the package that originally delivered this file does not contain the attribute `original_name`, add that attribute in the updated package. Set the value of the attribute to the name of the originating package, followed by a colon and the original path to the file without a leading `/`.

  Once the `original_name` attribute is present on an editable file, do not change the attribute value. This value acts as a unique identifier for all moves going forward so that the user's content is properly preserved regardless of the number of versions skipped on an update.

Moving Unpackaged Contents on Directory Removal or Rename

Normally, unpackaged contents are salvaged when the containing directory is removed, because the last reference to it disappears.

When a directory changes names, IPS treats this as the removal of the old directory and the creation of a new one. Any editable files that are still in the directory when the directory is renamed or removed are salvaged.

If the old directory has unpackaged content such as log files that should be moved to the new directory, use the `salvage-from` attribute on the new directory. For example, if `pkgA` renames a directory from `/opt/olddata/log` to `/opt/newdata/log`, then in the version of `pkgA` that makes this change, include the following attribute on the `dir` action that creates `/opt/newdata/log`:

```
salvage-from=opt/olddata/log
```

Any unpackaged contents of any time are migrated to the new location.

The `salvage-from` attribute is discussed again in "Delivering Directories To Be Shared Across Boot Environments" on page 87.

Delivering Multiple Implementations of an Application

You might want to deliver multiple implementations of a given application with characteristics such as the following:

- All implementations are available in the image.
- One of the implementations is specified as the preferred implementation.
- The preferred implementation has symbolic links to its binaries installed to a common directory such as `/usr/bin` for ease of discovery.
- The administrator can change the preferred implementation as required, without adding or removing any packages.

One example of delivering multiple implementations of an application is GCC. Oracle Solaris provides multiple versions of GCC, with each version in its own package, and `/usr/bin/gcc` points to the preferred version.

IPS uses mediated links to manage multiple implementations of an application in a single image. A *mediated link* is a symbolic link that is controlled by the `pkg set-mediator` and `pkg unset-mediator` commands. The `link` actions in the packages that deliver different implementations of an application are said to participate in a *mediation*. The `pkg mediator` command lists the mediations in the image. See the `pkg(1)` man page for information about the mediator commands.

The following attributes can be set on `link` actions to control how mediated links are delivered:

`mediator`
: Specifies the entry in the mediation namespace shared by all path names participating in a given mediation group (for example `python`).

  Link mediation can be performed based on `mediator-version` and `mediator-implementation`. All mediated links for a given path name must specify the same `mediator`. However, not all mediator versions and implementations need to provide a link at a given path. If a mediation does not provide a link, then the link is removed when that mediation is selected.

  A mediator, in combination with a specific version and/or implementation represents a *mediation* that can be selected for use by the packaging system.

`mediator-version`
: Specifies the version (expressed as a dot-separated sequence of non-negative integers) of the interface described by the `mediator` attribute. This attribute is required if `mediator` is specified and `mediator-implementation` is not specified. A local system administrator can explicitly set the version to use. The value specified generally should match the version of the package that is delivering the link. For example, `runtime/python-26` should use `mediator-version=2.6`, although this is not required.

`mediator-implementation`
: Specifies the implementation of the mediator. The attribute can be specified in addition to or instead of the `mediator-version` attribute. Implementation strings are not considered to be ordered. A string is arbitrarily selected by `pkg`(5) if not explicitly specified by a system administrator.

  The value of `mediator-implementation` can be a string of arbitrary length composed of alphanumeric characters and spaces. If the implementation itself can be or is versioned, then the version should be specified at the end of the string, after an `@` symbol. The version is a dot-separated sequence of non-negative integers. If multiple versions of an implementation exist, the default behavior is to select the implementation with the highest version.

  If only one instance of an implementation-mediation link at a particular path is installed on a system, then that one is chosen automatically. If future links at the path are installed, the link will not be switched unless a vendor, site, or local override applies, or if one of the links is version-mediated.

`mediator-priority`
: When resolving conflicts in mediated links, `pkg`(5) chooses the link with the greatest value of `mediator-version` if possible. If that is not possible, `pkg`(5) chooses the link based on `mediator-implementation`. The `mediator-priority` attribute is used to specify an override for the normal conflict resolution process. If the `mediator-priority` attribute is not specified, the default mediator selection logic is applied.

  The `mediator-priority` attribute can have one of the following values:

`vendor`	The link is preferred over those that do not have a `mediator-priority` specified.
`site`	The link is preferred over those that have a value of `vendor` or that do not have a `mediator-priority` specified.

  A local system administrator can override the selection logic described above.

The following two excerpts from sample manifests participate in a mediation for the link `/usr/bin/myapp`. Implementation 1 is version 5.8.4:

```
set name=pkg.fmri value=pkg://test/myapp-impl-1@1.0,5.11:20120721T035233Z
file path=usr/myapp/5.8.4/bin/myapp group=sys mode=0755 owner=root
link path=usr/bin/myapp target=usr/myapp/5.8.4/bin/myapp mediator=myapp mediator-version=5.8.4
```

Implementation 2 is version 5.12:

```
set name=pkg.fmri value=pkg://test/myapp-impl-2@1.0,5.11:20120721T035239Z
file path=usr/myapp/5.12/bin/myapp group=sys mode=0755 owner=root
link path=usr/bin/myapp target=usr/myapp/5.12/bin/myapp mediator=myapp mediator-version=5.12
```

Both of these packages can be installed in the same image:

```
$ pkg list myapp-impl-1 myapp-impl-2
NAME (PUBLISHER)                  VERSION      IFO
myapp-impl-1                      1.0          i--
myapp-impl-2                      1.0          i--
```

Use the pkg mediator command to see the mediations in use:

```
$ pkg mediator
MEDIATOR VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp    local     5.12    system
$ ls -al usr/bin/myapp
lrwxrwxrwx 1 root sys 23 Jul 21 16:58 usr/bin/myapp -> usr/myapp/5.12/bin/myapp
```

Use the pkg search command to see which other packages participate in the myapp mediation:

```
$ pkg search -ro path,target,mediator,mediator-version,pkg.shortfmri ::mediator:myapp
PATH          TARGET                    MEDIATOR MEDIATOR-VERSION PKG.SHORTFMRI
usr/bin/myapp usr/myapp/5.12/bin/myapp  myapp    5.12             pkg:/myapp-impl-2@1.0
usr/bin/myapp usr/myapp/5.8.4/bin/myapp myapp    5.8.4            pkg:/myapp-impl-1@1.0
```

Use the pkg set-mediator command to change the mediation. The following example changes which version of myapp is the preferred version:

```
# pkg set-mediator -V 5.8.4 myapp
            Packages to update: 2
           Mediators to change: 1
       Create boot environment: No
Create backup boot environment: No

PHASE                                    ITEMS
Removing old actions                       2/2
Updating modified actions                  2/2
Updating image state                      Done
Creating fast lookup database             Done
Reading search index                      Done
Updating search index                      2/2
# ls -al usr/bin/myapp
lrwxrwxrwx  1 root    sys    24 Jul 21 17:02 usr/bin/myapp -> usr/myapp/5.8.4/bin/myapp
```

Delivering Directories To Be Shared Across Boot Environments

Some files delivered to a boot environments (BE) must be shared across BEs to preserve normal system operation in an environment with multiple BEs. In general, IPS does not support delivery of content that is shared across BEs. Such shared content updated in one BE might not meet the requirements of other BEs. This section describes how you can deliver content that is shared across BEs.

The following directories are already shared across BEs by IPS:

```
/var/audit
/var/cores
/var/crash
/var/mail
```

Within each BE, these directories are symbolic links to the following shared directories:

```
/var/share/audit
/var/share/cores
/var/share/crash
/var/share/mail
```

These shared directories are in the `VARSHARE` dataset, which is a shared dataset mounted at `/var/share`.

If other data needs to be shared across BEs but is delivered in an IPS package as unshared, administrators are likely to place such data on separate datasets or on remote file servers. However, creating per-directory datasets means creating many datasets per zone, which is not desirable.

Instead, use the following procedure to create a package that delivers a shared dataset, or to modify a package to share content that was previously delivered as unshared. IPS supports update from an older version of a package that did not share content to a newer version of the same package that does share content across BEs.

▼ How To Deliver Content to Shared Directories

This procedure describes how to design a package that must deliver content that is shared across BEs.

To share data across BEs, use a shared dataset, mounted into the BE during boot, with symbolic links from locations inside the BE pointing into that shared dataset. Inside the BE, deliver basic directory structure to a staging directory. Provide an SMF service that moves the content staged inside the BE to the shared dataset during boot, and provide an actuator to require a reboot.

1 **Deliver shared content to a staging area in the BE.**

   a. **Deliver a staging area.**

      In your package, deliver a staging area for shared content. For example, you might deliver a directory named `.migrate`.

   b. **Deliver the shared structure.**

      Deliver subdirectories into the `.migrate` directory that provide the directory structure you want in the shared dataset.

c. **Deliver the shared files.**

   Deliver files to the directory structure in the staging area as needed. Other file system objects such as links cannot be shared.

If the content delivered to the staging area was previously delivered as unshared content, use a `salvage-from` attribute on the new `dir` or `file` action. In the following example, content that was previously delivered to `/opt/myapplication/logs` will now be delivered to a dataset that is shared across BEs. The staging area for this shared dataset is `/opt/.migrate`.

The following action was previously delivered:

```
dir path=opt/myapplication/logs owner=daemon group=daemon mode=0755
```

The following action is the new action for the directory that will be shared:

```
dir path=opt/.migrate/myapplication/logs owner=daemon group=daemon \
    mode=0755 reboot-needed=true salvage-from=/opt/myapplication/logs
```

The `salvage-from` attribute is also discussed in "Moving Unpackaged Contents on Directory Removal or Rename" on page 84.

2 **Provide a script to move content into the shared dataset.**

During boot, a script can be run as part of an SMF method script to move file content from the staging directory into the shared dataset. This script must do the following steps:

a. **Create the shared dataset.**

   The following command in the SMF method script creates the dataset `rpool/OPTSHARE` mounted at `/opt/share`. This dataset could also be used by other shared content from `/opt`. The script should use `zfs list` to test whether the dataset already exists.

```
zfs create -o mountpoint=/opt/share rpool/OPTSHARE
```

b. **Create the shared directory structure.**

   In the shared dataset, recreate any parts of the directory structure defined under the staging directory in the BE that do not already exist.

c. **Move file content.**

   Move the file content from the staging directory to the shared dataset.

3 **Deliver a symbolic link from the BE to the shared directory.**

The following action creates a symbolic link from the previously packaged directory to the shared directory in `/opt/share` that will be created by the script when the system reboots:

```
link path=opt/myapplication/logs target=../../opt/share/myapplication/logs
```

4 **Add an actuator to require a reboot.**

A `reboot-needed` actuator is required for these directory entries in order to properly support updates of Immutable Zones mentioned in "Software Self-Assembly" on page 13. Immutable Zones boot as far as the `svc:/milestone/self-assembly-complete:default` milestone in read/write mode if self-assembly is required, before rebooting read-only. See the `file-mac-profile` property in the `zonecfg`(1M) man page for more information.

On reboot, the SMF service moves any new and salvaged directory content into the shared dataset. The symbolic links from `/opt/myapplication` point into that shared dataset.

◆ ◆ ◆ CHAPTER 9

Signing IPS Packages

The ability to validate that the software installed on the user's machine is actually as originally specified by the publisher is an important feature of IPS. This ability to validate the installed system is key for both the user and the support engineering staff.

Signature policies can be set for the image or for specific publishers. Policies include ignoring signatures, verifying existing signatures, requiring signatures, and requiring specific common names in the chain of trust.

This chapter describes IPS package signing and how developers and quality assurance organizations can sign either new packages or existing, already signed packages.

Signing Package Manifests

IPS package manifests can be signed, with the signatures becoming part of the manifest.

Defining Signature Actions

Signatures are represented as actions just as all other manifest content is represented as actions. Since manifests contain all the package metadata (such as file permissions, ownership, and content hashes), a signature action that validates that the manifest has not be altered since it was published is an important part of system validation.

The `signature` actions form a tree that includes the delivered binaries such that complete verification of the installed software is possible.

In addition to validation, signatures can also be used to indicate approval by other organizations or parties. For example, the internal QA organization could sign manifests of packages once the packages are qualified for production use. Such approvals could be required for installation.

A manifest can have multiple independent signatures. Signatures can be added or removed without invalidating other signatures that are present. This feature facilitates production handoffs, with signatures used along the path to indicate completion along the way. Subsequent steps can optionally remove previous signatures at any time.

A `signature` action has the following form:

```
signature hash_of_certificate algorithm=signature_algorithm \
    value=signature_value \
    chain="hashes_of_certificates_needed_to_validate_primary_certificate" \
    version=pkg_version_of_signature
```

The payload and `chain` attributes represent the packaging hash of the PEM (Privacy Enhanced Mail) files, containing the x.509 certificates which can be retrieved from the originating repository. The payload certificate is the certificate that verifies the value in `value`. The `value` is the signed hash of the message text of the manifest, prepared as discussed below.

The other certificates presented need to form a certificate path that leads from the payload certificate to the trust anchors.

Two types of signature algorithms are supported:

RSA	The first type of signature algorithm is the RSA group of algorithms. An example of an RSA signature algorithm is `rsa-sha256`. The string after the hyphen (`sha256` in this example) specifies the hash algorithm to use to change the message text into a single value the RSA algorithm can use.
Hash only	The second type of signature algorithm is compute the hash only. This type of algorithm exists primarily for testing and process verification purposes and presents the hash as the signature value. A signature action of this type is indicated by the lack of a payload certificate hash. This type of signature action is verified if the image is configured to check signatures. However, its presence does not count as a signature if signatures are required. The following example shows a hash only signature action:

```
signature algorithm=hash_algorithm value=hash \
    version=pkg_version_of_signature
```

Publishing Signed Package Manifests

Publishing a signed manifest is a two step process. This process leaves the package intact, including its time stamp.

1. Publish the package unsigned to a repository.
2. Update the package in place, using the `pkgsign` command to append a signature action to the manifest in the repository.

This process enables a signature action to be added by someone other than the publisher without invalidating the original publisher's signature. For example, the QA department of a company might want to sign all packages that are installed internally to indicate they have been approved for use, but not republish the packages, since republishing would create a new time stamp and invalidate the signature of the original publisher.

Note that using the `pkgsign` command is the only way to publish a signed package. If you publish a package that already contains a signature, that signature is removed and a warning is emitted. The `pkgsign(1)` man page contains examples of how to use the `pkgsign` command.

Signature actions with variants are ignored. Therefore, performing a `pkgmerge` on a pair of manifests invalidates any signatures that were previously applied.

Note – Signing the package should be the last step of the package development before the package is tested.

Troubleshooting Signed Packages

The `pkgsign` tool does not perform all possible checks for its inputs when signing packages. Therefore, it is important to check signed packages to ensure that they can be properly installed after being signed.

This section shows errors that can occur when attempting to install or update a signed package and provides explanations of the errors and solutions to the problems.

A signed package can fail to install or update for reasons that are unique to signed packages. For example, if the signature of a package fails to verify, or if the chain of trust cannot be verified or anchored to a trusted certificate, the package fails to install.

When installing signed packages, the following image and publisher properties influence the checks that are performed on packages:

Image properties

- `signature-policy`
- `signature-required-names`
- `trust-anchor-directory`

Publisher properties

- `signature-policy`
- `signature-required-names`

See the `pkg(1)` man page for further information about these properties and their values.

Chain Certificate Not Found

The following error occurs when a certificate in the chain of trust is missing or otherwise erroneous.

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta3/emailAddress=cs1_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T184152Z
```

In this example, there were three certificates in the chain of trust when the package was signed. The chain of trust was rooted in the trust anchor, a certificate named `ta3`. The `ta3` certificate signed a chain certificate named `ch1_ta3`, and `ch1_ta3` signed a code signing certificate named `cs1_ch1_ta3`.

When the `pkg` command tried to install the package, it was able to locate the code signing certificate, `cs1_ch1_ta3`, but could not locate the chain certificate, `ch1_ta3`, so the chain of trust could not be established.

The most common cause of this problem is failing to provide the correct certificates to the `-i` option of `pkgsign`.

Authorized Certificate Not Found

The following error is similar to the error shown in the previous example but the cause is different.

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_cs8_ch1_ta3/emailAddress=cs1_cs8_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs8_ch1_ta3/emailAddress=cs8_ch1_ta3
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T201101Z
```

In this case, the package was signed using the `cs1_cs8_ch1_ta3` certificate, which was signed by the `cs8_ch1_ta3` certificate.

The problem is that the `cs8_ch1_ta3` certificate was not authorized to sign other certificates. Specifically, the `cs8_ch1_ta3` certificate had the `basicConstraints` extension set to `CA:false` and marked critical.

When the `pkg` command verifies the chain of trust, it does not find a certificate that is allowed to sign the `cs1_cs8_ch1_ta3` certificate. Since the chain of trust cannot be verified from the leaf to the root, the `pkg` command prevents the package from being installed.

Untrusted Self-Signed Certificate

The following error occurs when a chain of trust ends in a self-signed certificate that is not trusted by the system.

```
pkg install: Chain was rooted in an untrusted self-signed certificate.
The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T185335Z
```

When you create a chain of certificates using OpenSSL for testing, the root certificate is usually self-signed, since there is little reason to have an outside company verify a certificate that is only used for testing.

In a test situation, there are two solutions:

- The first solution is to add the self-signed certificate that is the root of the chain of trust into `/etc/certs/CA` and refresh the `system/ca-certificates` service. This mirrors the likely situation customers will encounter where a production package is signed with a certificate that is ultimately rooted in a certificate that is delivered with the operating system as a trust anchor.
- The second solution is to approve the self-signed certificate for the publisher that offers the package for testing by using the `--approve-ca-cert` option with the `pkg set-publisher` command.

Signature Value Does Not Match Expected Value

The following error occurs when the value on the `signature` action could not be verified using the certificate that the action claims was paired with the key used to sign the package.

```
pkg install: A signature in pkg://test/example_pkg@1.0,5.11-0:20110919T195801Z
could not be verified for this reason:
The signature value did not match the expected value. Res: 0
The signature’s hash is 0ce15c572961b7a0413b8390c90b7cac18ee9010
```

There are two possible causes for an error like this:

- The first possible cause is that the package has been changed since it was signed. This is unlikely but is possible if the package manifest has been hand edited since signing. Without manual intervention, the package should not have changed since it was signed because `pkgsend` strips existing `signature` actions during publication because the old signature is invalid when the package gets a new time stamp.
- The second, more likely cause is that the key and certificate used to the sign the package were not a matched pair. If the certificate given to the `-c` option of `pkgsign` was not created with the key given to the `-k` option of `pkgsign`, the package is signed, but its signature will not be verified.

Unknown Critical Extension

The following error occurs when a certificate in the chain of trust uses a critical extension that pkg does not understand.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs2_ch1_ta3/emailAddress=cs2_ch1_ta3
could not be verified because it uses a critical extension that pkg5 cannot
handle yet. Extension name:issuerAltName
Extension value:<EMPTY>
```

Until pkg learns how to process that critical extension, the only solution is to regenerate the certificate without the problematic critical extension.

Unknown Extension Value

The following error is similar to the previous error except that the problem is not with an unfamiliar critical extension but with a value that pkg does not understand for an extension that pkg does understand.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs5_ch1_ta3/emailAddress=cs5_ch1_ta3
could not be verified because it has an extension with a value that pkg(5)
does not understand.
Extension name:keyUsage
Extension value:Encipher Only
```

In this case, pkg understands the keyUsage extension, but does not understand the value Encipher Only. The error looks the same whether the extension in question is critical or not.

The solution, until pkg learns about the value in question, is to remove the value from the extension, or remove the extension entirely.

Unauthorized Use of Certificate

The following error occurs when a certificate has been used for a purpose for which it was not authorized.

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
could not be verified because it has been used inappropriately.
The way it is used means that the value for extension keyUsage must include
’DIGITAL SIGNATURE’ but the value was ’Certificate Sign, CRL Sign’.
```

In this case, the certificate `ch1_ta3` has been used to sign the package. The `keyUsage` extension of the certificate means that the certificate is only valid to sign other certificates and CRLs (Certificate Revocation Lists).

Unexpected Hash Value

The following error indicates that the certificate has been changed since it was last retrieved from the publisher.

```
pkg install: Certificate
/tmp/ips.test.7149/0/image0/var/pkg/publisher/test/certs/0ce15c572961b7a0413b8390c90b7cac18ee9010
has been modified on disk. Its hash value is not what was expected.
```

The certificate at the provided path is used to verify the package being installed, but the hash of the contents on disk do not match what the signature action thought they should be.

The simple solution is to remove the certificate and allow `pkg` to download the certificate again.

Revoked Certificate

The following error indicates the certificate in question, which was in the chain of trust for the package to be installed, was revoked by the issuer of the certificate.

```
pkg install: This certificate was revoked:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta4/emailAddress=cs1_ch1_ta4
for this reason: None
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T205539Z
```

CHAPTER 10

Handling Non-Global Zones

Developing packages that work consistently with zones usually involves little or no additional work. This chapter describes how IPS handles zones and discusses those cases where packaging needs to account for non-global zones.

Packaging Considerations for Non-Global Zones

When considering zones and packaging, two questions need to be answered:

- Does anything in my package have an interface that crosses the boundary between the global zone and non-global zones?
- How much of the package should be installed in the non-global zone?

Does the Package Cross the Global, Non-Global Zone Boundary?

If `pkgA` delivers both kernel and userland functionality, and both sides of that interface must be updated accordingly, then whenever `pkgA` is updated in a non-global zone, `pkgA` must also be updated in any other zones where `pkgA` is installed.

To ensure this update is done correctly, use a `parent` dependency in `pkgA`. If a single package delivers both sides of the interface, then a `parent` dependency on `feature/package/dependency/self` ensures that the global zone and the non-global zones contain the same version of the package, preventing version skew across the interface.

The `parent` dependency also ensures that if the package is in a non-global zone, then it is also present in the global zone.

If the interface spans multiple packages, then the package that contains the non-global zone side of the interface must contain a `parent` dependency on the package that delivers the global zone side of the interface. The `parent` dependency is also discussed in "Dependency Types" on page 57.

How Much of a Package Should Be Installed in a Non-Global Zone?

If all of a package should be installed when the package is being installed in a non-global zone, then nothing needs to be done to the package to enable it to function properly. For consumers of the package, though, it can be reassuring to see that the package author properly considered zone installation and decided that this package can function in a zone. For that reason, you should explicitly state that the package functions in both global and non-global zones. To do this, add the following action to the manifest:

```
set name=variant.opensolaris.zone value=global value=nonglobal
```

If no content in the package can be installed in a non-global zone (for example a package that only delivers kernel modules or drivers), then the package should specify that it cannot be installed in a non-global zone. To do this, add the following action to the manifest:

```
set name=variant.opensolaris.zone value=global
```

If some but not all of the content in the package can be installed in a non-global zone, then take the following steps:

1. Use the following `set` action to state that the package can be installed in both global and non-global zones:

   ```
   set name=variant.opensolaris.zone value=global value=nonglobal
   ```
2. Identify the actions that are relevant only in the global zone or only in a non-global zone. Assign the following attribute to actions that are relevant only in the global zone:

   ```
   variant.opensolaris.zone=global
   ```

   Assign the following attribute to actions that are relevant only in a non-global zone:

   ```
   zone:variant.opensolaris.zone=nonglobal
   ```

If a package has a `parent` dependency or has pieces that are different in global and non-global zones, test to ensure that the package works as expected in a non-global zone as well as in the global zone.

If the package has a `parent` dependency on itself, then the global zone must configure the repository that delivers the package as one of its origins. Install the package in the global zone first, and then in the non-global zone for testing.

Troubleshooting Package Installations in Non-Global Zones

This section discusses problems that users might encounter when attempting to install a package in a non-global zone.

Packages that Have Parent Dependencies on Themselves

If you encounter a problem installing a package in a non-global zone, ensure that the following services are online in the global zone:

```
svc:/application/pkg/zones-proxyd:default
svc:/application/pkg/system-repository:default
```

Ensure that the following service is online in the non-global zone:

```
svc:/application/pkg/zones-proxy-client:default
```

These three services provide publisher configuration to the non-global zone and a communication channel that the non-global zone can use to make requests to the repositories assigned to the system publishers served from the global zone.

You cannot update the package in the non-global zone, since it has a parent dependency on itself. Initiate the update from the global zone; `pkg` updates the non-global zone along with the global zone.

Once the package is installed in the non-global zone, test the functionality of the package.

Packages that Do Not Have Parent Dependencies on Themselves

If the package does not have a `parent` dependency on itself, then you do not need to configure the publisher in the global zone, and you should not install the package in the global zone. Updating the package in the global zone will not update the package in the non-global zone. In this case, updating the package in the global zone can cause unexpected results when testing the older non-global zone package.

The simplest solution in this situation is to make the publisher available to the non-global zone, and install and update the package from within the non-global zone.

If the zone cannot access the publisher's repositories, configuring the publisher in the global zone enables the `zones-proxy-client` and `system-repository` services to proxy access to the publisher for the non-global zone. Then install and update the package in the non-global zone.

CHAPTER 11

Modifying Published Packages

Occasionally, you might need to modify packages that you did not produce. For example, you might need to override attributes, replace a portion of the package with an internal implementation, or remove binaries that are not permitted on your systems.

This chapter describes how you can modify existing packages for local conditions.

Republishing Packages

IPS enables you to easily republish an existing package with your modifications, even if you did not originally publish the package. You can also republish new versions of the modified package so that `pkg update` continues to work as users expect. Modified packages install and update correctly in the image.

Of course, running a system with a modified package could adversely affect your support if any relationship is suspected between observed problems and the modified package.

Use the following steps to modify and republish a package:

1. Use `pkgrecv(1)` to download the package to be republished in a raw format to a specified directory. All of the files are named by their hash value, and the manifest is named `manifest`. Remember to set any required proxy configuration in the `http_proxy` environment variable.
2. Use `pkgmogrify(1)` to make the necessary modifications to the manifest. Remove any time stamp from the internal package FMRI to prevent confusion during publication.

   If changes are significant, use `pkglint(1)` to verify the resulting package.
3. Use `pkgsend(1)` to republish the package. Note that this republication strips the package of any signatures that are present and ignores any time stamp specified by `pkg.fmri`. To prevent a warning message, remove signature actions in the `pkgmogrify` step.

If you do not have permission to publish to the original source of the package, use pkgrepo(1) to create a repository, and then use the following command to set the new publisher ahead of the original publisher in the publisher search order:

```
# pkg set-publisher --search-before=original_publisher  new_publisher
```

4. If necessary, use pkgsign(1) to sign the package. To prevent client caching issues, sign the package before you install the package, even for testing.

Changing Package Metadata

In the following example, the original `pkg.summary` value is changed to be "IPS has lots of features." The package is downloaded using the `--raw` option of `pkgrecv`. By default, only the newest version of the package is downloaded. The package is then republished to a new repository.

```
$ mkdir republish; cd republish
$ pkgrecv -d . --raw -s http://pkg.oracle.com/solaris/release package/pkg
$ cd package* # The package name contains a '/' and is url-encoded.
$ cd *
$ cat > fix-pkg
# Change the value of pkg.summary
<transform set name=pkg.summary -> edit value '.*' "IPS has lots of features">
# Delete any signature actions
<transform signature -> drop>
# Remove the time stamp from the fmri so that the new package gets a new time stamp
<transform set name=pkg.fmri -> edit value ":20.+" "">
^D
$ pkgmogrify manifest fix-pkg > new-manifest
$ pkgrepo create ./mypkg
$ pkgsend -s ./mypkg publish -d . new-manifest
```

Changing Package Publisher

Another common use case is to republish packages under a new publisher name. One case when this is useful is to consolidate packages from multiple repositories into a single repository. For example, you might want to consolidate packages from repositories of several different development teams into a single repository for integration testing.

To republish under a new publisher name, use the `pkgrecv`, `pkgmogrify`, `pkgrepo`, and `pkgsend` steps shown in the previous example.

The following sample transform changes the publisher to `mypublisher`:

```
<transform set name=pkg.fmri -> edit value pkg://[^/]+/ pkg://mypublisher/>
```

You can use a simple shell script to iterate over all packages in the repository. Use the output from `pkgrecv --newest` to process only the newest packages from the repository.

The following script saves the above transform in a file named `change-pub.mog`, and then republishes from `development-repo` to a new repository `mypublisher`, changing the package publisher along the way:

```
#!/usr/bin/ksh93
pkgrepo create mypublisher
pkgrepo -s mypublisher set publisher/prefix=mypublisher
mkdir incoming
for package in $(pkgrecv -s ./development-repo --newest); do
    pkgrecv -s development-repo -d incoming --raw $package
done
for pdir in incoming/*/* ; do
    pkgmogrify $pdir/manifest change-pub.mog > $pdir/manifest.newpub
    pkgsend -s mypublisher publish -d $pdir $pdir/manifest.newpub
done
```

This script could be modified to do tasks such as select only certain packages, make additional changes to the versioning scheme of the packages, and show progress as it republishes each package.

◆ ◆ ◆ APPENDIX A

Classifying Packages

This appendix shows package information classification scheme definitions.

Assigning Classifications

The Package Manager GUI uses the `info.classification` package attribute, with scheme `org.opensolaris.category.2008`, to display packages by category. Users can also use the `pkg search` command to display packages that have a given classification.

Use a `set` action to assign a classification to a package, as shown in the following example:

```
set name=info.classification \
    value="org.opensolaris.category.2008:System/Administration and Configuration"
```

The category and subcategory are separated by a forward slash character. Spaces in the attribute value require quoting.

A package can have more than one classification, as shown in the following example:

```
set name=info.classification \
    value="org.opensolaris.category.2008:Meta Packages/Group Packages" \
    value="org.opensolaris.category.2008:Web Services/Application and Web Servers"
```

Classification Values

The following category and subcategory values are defined:

Meta Packages

- Group Packages
- Incorporations

Applications

- Accessories
- Configuration and Preferences
- Games
- Graphics and Imaging
- Internet
- Office
- Panels and Applets
- Plug-ins and Run-times
- Sound and Video
- System Utilities
- Universal Access

Desktop (GNOME)

- Documentation
- File Managers
- Libraries
- Localizations
- Scripts
- Sessions
- Theming
- Trusted Extensions
- Window Managers

Development

- C
- C++
- Databases
- Distribution Tools
- Editors
- Fortran
- GNOME and GTK+
- GNU
- High Performance Computing
- Java
- Objective C
- Other Languages
- PHP
- Perl
- Python
- Ruby

- Source Code Management
- Suites
- System
- X11

Drivers

- Display
- Media
- Networking
- Other Peripherals
- Ports
- Storage

System

- Administration and Configuration
- Core
- Databases
- Enterprise Management
- File System
- Fonts
- Hardware
- Internationalization
- Libraries
- Localizations
- Media
- Multimedia Libraries
- Packaging
- Printing
- Security
- Services
- Shells
- Software Management
- Text Tools
- Trusted
- Virtualization
- X11

Web Services

- Application and Web Servers
- Communications

◆ ◆ ◆ APPENDIX B

How IPS Is Used To Package the Oracle Solaris OS

This appendix describes how Oracle uses IPS features to package the Oracle Solaris OS, and how the various dependency types are used to define working package sets for the OS.

This appendix provides another concrete example of using IPS to manage a complex set of software.

Oracle Solaris Package Versioning

"Package Identifier: FMRI" on page 17 described the `pkg.fmri` attribute and the different components of the version field, including how the version field can be used to support different models of software development. This section explains how the Oracle Solaris OS uses the version field, and provides insight into the reasons that a fine-grained versioning scheme can be useful. In your packages, you do not need to follow the same versioning scheme that the Oracle Solaris OS uses.

The meaning of each part of the version string in the following sample package FMRI is given below:

```
pkg://solaris/system/library@0.5.11,5.11-0.175.1.0.0.2.1:20120919T082311Z
```

`0.5.11`
: Component version. For packages that are part of the Oracle Solaris OS, this is the OS `major.minor` version. For other packages, this is the upstream version. For example, the component version of the following Apache Web Server package is 2.2.22:

```
pkg:/web/server/apache-22@2.2.22,5.11-0.175.1.0.0.2.1:20120919T122323Z
```

`5.11`
: Build version. This is used to define the OS release that this package was built for. The build version should always be 5.11 for packages created for Oracle Solaris 11.

`0.175.1.0.0.2.1`
Branch version. Oracle Solaris packages show the following information in the branch version portion of the version string of a package FMRI:

`0.175`	Major release number. The major or marketing development release build number. In this example, `0.175` indicates Oracle Solaris 11.
`1`	Update release number. The update release number for this Oracle Solaris release. The update value is 0 for the first customer shipment of an Oracle Solaris release, 1 for the first update of that release, 2 for the second update of that release, and so forth. In this example, `1` indicates Oracle Solaris 11.1.
`0`	SRU number. The Support Repository Update (SRU) number for this update release. SRUs include only bug fixes; they do not include new features. The Oracle Support Repository is available only to systems under a support contract.
`0`	Reserved. This field is not currently used for Oracle Solaris packages.
`2`	SRU build number. The build number of the SRU, or the respin number for the major release.
`1`	Nightly build number. The build number for the individual nightly builds.

`20120919T082311Z`
Time stamp. The time stamp was defined when the package was published.

Oracle Solaris Incorporation Packages

Oracle Solaris is delivered by a set of packages, with each group of packages constrained by an incorporation.

Each incorporation roughly represents the organization that developed each group of packages, though there are some cross-incorporation dependencies within the packages themselves. The following incorporation packages are in Oracle Solaris (`pkg list *incorporation`):

```
pkg:/consolidation/SunVTS/SunVTS-incorporation
pkg:/consolidation/X/X-incorporation
pkg:/consolidation/admin/admin-incorporation
pkg:/consolidation/cacao/cacao-incorporation
pkg:/consolidation/cde/cde-incorporation
pkg:/consolidation/cns/cns-incorporation
pkg:/consolidation/dbtg/dbtg-incorporation
pkg:/consolidation/desktop/desktop-incorporation
pkg:/consolidation/desktop/gnome-incorporation
pkg:/consolidation/gfx/gfx-incorporation
pkg:/consolidation/install/install-incorporation
pkg:/consolidation/ips/ips-incorporation
pkg:/consolidation/java/java-incorporation
pkg:/consolidation/jdmk/jdmk-incorporation
pkg:/consolidation/l10n/l10n-incorporation
```

```
pkg:/consolidation/ldoms/ldoms-incorporation
pkg:/consolidation/man/man-incorporation
pkg:/consolidation/nspg/nspg-incorporation
pkg:/consolidation/nvidia/nvidia-incorporation
pkg:/consolidation/osnet/osnet-incorporation
pkg:/consolidation/sfw/sfw-incorporation
pkg:/consolidation/sic_team/sic_team-incorporation
pkg:/consolidation/solaris_re/solaris_re-incorporation
pkg:/consolidation/sunpro/sunpro-incorporation
pkg:/consolidation/ub_javavm/ub_javavm-incorporation
pkg:/consolidation/userland/userland-incorporation
pkg:/consolidation/vpanels/vpanels-incorporation
pkg:/consolidation/xvm/xvm-incorporation
```

Each of these incorporations includes the following information:

- Package metadata.
- Dependencies of type `incorporate`, sometimes with `variant.arch` variants to denote dependencies that are specific to a given architecture. See “`incorporate` Dependency” on page 60 and “Mutually Exclusive Software Components” on page 65 for more information about `incorporate` dependencies and `variant.arch` variants.
- A `license` action to ensure that a license is displayed when the incorporation is installed. See “License Actions” on page 28 for more information about `license` actions.

Each package delivered on the system contains a `require` dependency on one of these incorporations. See “`require` Dependency” on page 57 for more information.

Oracle Solaris also includes a special incorporation named `entire`. The `entire` incorporation constrains all other incorporations to the same build by including both `require` and `incorporate` dependencies on each incorporation package. In this way, the `entire` incorporation defines a software surface such that all packages are upgraded as a single group.

Relaxing Dependency Constraints

Some of the incorporations listed above use `facet.version-lock.*` facets to enable the administrator to use the `pkg change-facet` command to relax the constraint to the incorporation for the specified packages. See “Relaxing Constraints on Installable Package Versions” on page 62 for more information.

For example, the `pkg:/consolidation/userland/userland-incorporation` package contains the following `facet.version-lock.*` definitions:

```
..
depend type=incorporate \
    fmri=pkg:/library/python-2/subversion@1.6.16-0.175.0.0.0.2.537 \
    facet.version-lock.library/python-2/subversion=true
depend type=incorporate \
    fmri=pkg:/library/security/libassuan@2.0.1-0.175.0.0.0.2.537 \
    facet.version-lock.library/security/libassuan=true
depend type=incorporate \
    fmri=pkg:/library/security/openssl/openssl-fips-140@1.2-0.175.0.0.0.2.537 \
    facet.version-lock.library/security/openssl/openssl-fips-140=true
depend type=incorporate fmri=pkg:/mail/fetchmail@6.3.21-0.175.0.0.0.2.537 \
    facet.version-lock.mail/fetchmail=true
depend type=incorporate \
    fmri=pkg:/network/chat/ircii@0.2006.7.25-0.175.0.0.0.2.537 \
    facet.version-lock.network/chat/ircii=true
depend type=incorporate \
    fmri=pkg:/print/cups/filter/foomatic-db-engine@0.20080903-0.175.0.0.0.2.537 \
    facet.version-lock.print/cups/filter/foomatic-db-engine=true
depend type=incorporate \
    fmri=pkg:/print/filter/gutenprint@5.2.4-0.175.0.0.0.2.537 \
    facet.version-lock.print/filter/gutenprint=true
depend type=incorporate fmri=pkg:/runtime/erlang@12.2.5-0.175.0.0.0.2.537 \
facet.version-lock.runtime/erlang=true
    ..
```

The `entire` package also contains `version-lock` facets. In this case, the facets allow specified incorporations to be removed from the `entire` incorporation. However, this can result in a system that is not covered by support. Those packages should only be unlocked on advice from Oracle support personnel.

Oracle Solaris Group Packages

Oracle Solaris defines several group packages that contain `group` dependencies. See "`group` Dependency" on page 59 for more information about `group` dependencies. These group packages enable convenient installation of common sets of packages.

The following group packages are in Oracle Solaris (`pkg list -a group*`):

```
pkg:/group/feature/amp
pkg:/group/feature/developer-gnu
pkg:/group/feature/multi-user-desktop
pkg:/group/feature/storage-avs
pkg:/group/feature/storage-nas
pkg:/group/feature/storage-server
pkg:/group/feature/trusted-desktop
pkg:/group/system/solaris-auto-install
pkg:/group/system/solaris-desktop
pkg:/group/system/solaris-large-server
pkg:/group/system/solaris-small-server
```

The `solaris-small-server` group package is installed by the default AI manifest that is used to install non-global zones (`/usr/share/auto_install/manifest/zone_default.xml`). See `solaris(5)` for more information.

Attributes and Tags

This section describes general and Oracle Solaris action attributes and Oracle Solaris attribute tags.

Informational Attributes

The following attributes are not necessary for correct package installation, but having a shared convention reduces confusion between publishers and users.

`info.classification`
: See "Set Actions" on page 24 for information about the `info.classification` attribute. See a list of classifications in Appendix A, "Classifying Packages."

`info.keyword`
: A list of additional terms that should cause this package to be returned by a search.

`info.maintainer`
: A human readable string that describes the entity that provides the package. This string should be the name, or name and email of an individual, or the name of an organization.

`info.maintainer-url`
: A URL associated with the entity that provides the package.

`info.upstream`
: A human readable string that describes the entity that creates the software. This string should be the name, or name and email of an individual, or the name of an organization.

`info.upstream-url`
: A URL associated with the entity that creates the software delivered in the package.

`info.source-url`
: A URL to the source code bundle for the package, if appropriate.

`info.repository-url`
: A URL to the source code repository for the package, if appropriate.

`info.repository-changeset`
: A changeset ID for the version of the source code contained in `info.repository-url`.

Oracle Solaris Attributes

`org.opensolaris.arc-caseid`	One or more case identifiers (for example, PSARC/2008/190) associated with the ARC case (Architecture Review Committee) or cases associated with the component delivered by the package.

`org.opensolaris.smf.fmri`	One or more FMRIs that represent SMF services delivered by this package. These attributes are automatically generated by `pkgdepend` for packages that contain SMF service manifests. See the `pkgdepend(1)` man page.

Organization-Specific Attributes

To provide additional metadata for a package, use an organization-specific prefix on the attribute name. Organizations can use this method to provide additional metadata for packages developed in that organization or to amend the metadata of an existing package. To amend the metadata of an existing package, you must have control over the repository where the package is published. For example, a service organization might introduce an attribute named `service.example.com,support-level` or `com.example.service,support-level` to describe a level of support for a package and its contents.

Oracle Solaris Tags

`variant.opensolaris.zone`	Specifies which actions in a package can be installed in a non-global zone, in the global zone, or in either a non-global or the global zone. See Chapter 10, “Handling Non-Global Zones,” for more information.