mkpkg
A Software Packaging Tool

Carl Staelin, Hewlett-Packard Laboratories

mkpkg: A software packaging tool

Carl Staelin – Hewlett-Packard Laboratories

ABSTRACT

mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, *mkpkg* helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. *mkpkg* automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using *mkpkg*, a publisher can generate software packages for complex software such as TeX with only a few minutes effort.

mkpkg has been implemented on HP-UX using Tcl/Tk and provides both graphical and command line interfaces. It builds product-level packages for Software Distributor (SD-UX).

Introduction

Most end-users do not build programs from source code, but install software using binary installation packages. *mkpkg* helps software publishers develop those installation packages. Building my first complex installation package by hand took me a week, but with *mkpkg* I can build installation packages with roughly three minutes of effort.

mkpkg addresses a part of the software distribution channel that has been largely ignored. Most software distribution systems have focussed on defining the binary package format and the protocols for installing and de-installing software. Most software installation suites have made it very easy for end-users and system administrators to distribute and install software, but they have not addressed the problems of the software packager who is creating binary installation packages.

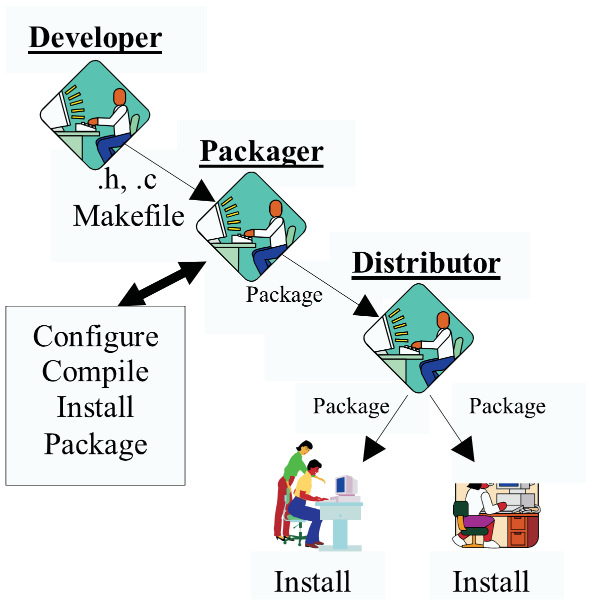

Figure 1: Software publishing process.

The software publishing process is illustrated in Figure 1 and includes several actors and steps: the software developer, the software publisher, the distributor, (sometimes the system administrator), and the end-user. The software developer creates the software. The packager is responsible for configuring, compiling, and packaging the software to create the binary installation package that the distributor delivers to the end-user. In some environments system administrators install and manage the software for end-users.

mkpkg helps software packagers create installation packages. Typically, the packager starts with source code that needs to be compiled and installed on the packager's computer. The packager tries to create an installation package that re-creates the installation on each end-user's computer.

Developing a binary software installation package, that is, creating a package that can be installed easily on a computers and have it work properly, is an important and difficult task. Most vendors have developed tools that can accept the descriptions of a software package and create an installation package, but developing those package descriptions is difficult. Package descriptions typically contain the elements listed in Table 1.

Each software installation tool has its own idiosyncrasies and requirements, but they all share these common elements. Many software installation tools also provide other elements, such as system specifications that define which hardware/OS types or versions may install the software.

mkpkg works in conjunction with software distribution tools, such as Software Distributor, by assembling all the elements required by the tool-specific packaging program, such as Software Distributor's swpackage. *mkpkg* is flexible and its back-end can be modified to create packages for different packaging tools. A Linux/RPM port is planned for the future.

Element	Description
title	Software package name
description	A text description of the package and its capabilities
manifest	A list of all the files contained in the package
dependencies	A list of all the other packages required for this package to operate correctly
customization scripts	A set of scripts that are executed on the user's machine during installation or de-installation of the software

Table 1: Elements in packages.

Software Distributor

Software Distributor (SD-UX) is a suite of software installation programs that satisfy the POSIX draft 1387.2 specification. Packagers use swpackage to transform a Package Specification File, binary files, installation scripts, and other files into a complete binary package. swcopy creates and manages repositories of installation packages, and swinstall actually installs software. Software Distributor is the software distribution mechanism for all Hewlett-Packard software for HP-UX and has versions that run on at least WindowsNT and Solaris.

Figure 2 demonstrates the four levels of software grouping in Software Distributor: bundle, product, subproduct, and fileset. A bundle is a collection of products and/or filesets that may be installed as a unit. Bundles were designed to provide customers with one single installation unit for purchased software products, such as the ANSI/C compiler. Bundles may be used to provide a logical grouping by function, such as "web server."

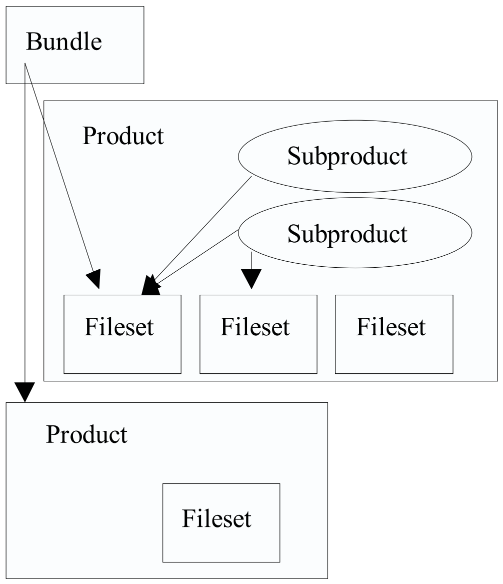

Figure 2: Four levels of software grouping.

The basic unit of software distribution is the product. A product may contain both subproducts and filesets. Subproducts contain filesets and are used to manage logical subsets of a single product. For example, the ANSI-C compiler product might have a subproduct for each language containing all the filesets needed to install the compiler with messages and documentation in the proper language.

Filesets are the atomic units of software distribution and contain a set of files and control scripts. SD-UX installs and configures individual filesets; filesets cannot be partially installed or configured.

Software Distributor has two levels of software distribution: bundle and product. The basic unit of distribution is the product. Software Distributor has several levels of software installation. The basic unit of installation is the fileset, but customers usually install software at the bundle or subproduct level.

mkpkg creates product packages, while mkbdl creates bundles. Since all filesets and subproducts are created as part of a product, we do not provide a separate tool for creating them.

Installation Tools

There are many methods for distributing binary installations. Each method has various strengths and weaknesses, but most commercial systems provide a similar level of basic operation. The largest UNIX vendors have each developed their own systems for distributing binary software: HP's HP-UX uses Software Distributor, Sun's Solaris uses pkgadd, Digital's OSF/1 uses setld, SGI's IRIX uses inst, and Linux uses RPM. Windows has two standards, InstallShield packages and self-extracting programs.

Each of the commercial systems offers basic services, such as installing and deinstalling packages atomically, tracking and managing inter-package dependencies, and executing scripts during software installation and deinstallation. Most of them also support a variety of more advanced features, such as versions, operating system and hardware dependencies, and interactive installation. For the most part these systems try to make it as easy as possible for system administrators to install software.

Tar

The simplest binary installation package is simply a tar file containing the software. Often such packages include a README file that includes installation instructions. For simple programs and packages, tar files are often sufficient. For more complex packages, much of the burden of correctly installing and configuring the software falls on the end user because the installation process for tar files is completely manual.

Occasionally, software publishers will include installation scripts as part of the tar file and the installation script will automatically install and configure the software for the user. One of the few publishers using this approach is Netscape, who includes an "nsinstall" script as part of the Netscape Communicator tar-file distribution.

RPM

The RedHat Package Manager (RPM) [2,3] was developed for the Linux environment and provides a very nice environment for installing and distributing software. Functionally, it is very similar to Software Distributor; it includes support for inter-package dependencies and control scripts that are executed during software installation.

Users may install software from a local depot or they may install from a remote server over the network. RPM is able to contain binaries for multiple platforms within a single package, and it can automatically install the correct binaries. Using RPM customers may determine which package installed a particular file, and what software is installed on the machine. Users may also uninstall packages.

RPM has some support for the packager, but it is missing some important features. RPM does not help the publisher develop the package manifest.

Software Configuration

Software configuration is one of the dirty little secrets of system administration. Software that is well configured works well in a broad variety of system configurations and causes few problems for system administrators. Poorly configured software can cause system administrators a great deal of aggravation.

Sometimes the difference between well-configured software and poorly configured software is a matter of tiny details, but there are a few general guidelines.

- Never compile paths into binaries.
- Separate executables, configuration files, and data or log files.
- Use human readable ASCII files for configuration information.
- Follow standard conventions for file and path names as much as possible.

System administrators frequently share file systems between systems, so executables and libraries will often be shared by many systems. However, administrators usually want each computer to have its own version of configuration files, but these may be on a read-only file system. In addition, data and log files are usually not shared between systems and usually must be mounted with read-write permissions. Software configurations must anticipate these kinds of configuration issues.

A fairly detailed set of configuration guidelines is published by the HP-UX Porting and Archive Centre [4].

Software Packaging Process

During software packaging, the publisher must prepare all the elements needed by the installation package. For many small packages, this is a very simple process, but for larger packages it can be quite difficult. *mkpkg* provides five services during the packaging process:

- Creates the manifest, the list of files to be installed
- Determines the dependencies of this package on other packages
- Develops the install/de-install scripts
- Gathers all the components, as listed in the manifest
- Assembles and produces the completed installation package

Create Manifest

The manifest is a list of all the files to be installed by the package. *mkpkg* can automatically determine which files were installed by the package on the publisher's machine. For small packages it is easy to determine which files belong to a given package, but manual techniques often miss files and make mistakes. For larger packages, such as X11R6 or TeX, it is usually difficult to identify all the files installed by the package and it is critical to include all the files that belong to the package.

Determine Dependencies

Many packages require the presence of other software in order to operate correctly. For example, if cvs uses rcs, then cvs depends on rcs. Packages can depend on other packages for many reasons, but executing programs and linking with shared libraries from other packages causes the two most common dependencies. Publishers are usually aware of the dependencies caused by executing programs, but often overlook shared library dependencies.

Develop Scripts

Typically, installation tools allow the publisher to add two kinds of scripts; those executed by the tool during installation and those executed during de-installation to erase all trace of the software. Also, some software packages require customized scripts to handle special configuration during the installation process. For example, many database systems require a special userid to be added to the system.

Writing these scripts is very difficult, but many actions can be specified in a general fashion. In order to simplify the development of scripts, the publisher simply specifies the desired results of executing the script, and *mkpkg* generates all the scripts needed for the package.

Gather Components

Once the package is specified, *mkpkg* gathers all the components, such as the customization scripts and installed files, and saves them in a temporary location. In the case for which multiple versions of a single package may be built (e.g., one version for statically linked binaries and another for dynamically linked binaries), the system may generate multiple copies of the system and save them in different locations.

Assemble Package

The last step is to assemble the installation package from the package configuration, customization scripts, and saved installation. *mkpkg* creates a Product Specification File (PSF) and all the automatically generated customization scripts and then uses swpackage to assemble and generate the completed package.

The PSF describes all the elements, options, and content of the installation package and is used by Software Distributor during package creation. Before *mkpkg* the PSF was nearly always generated manually.

Automation

Since many of the tasks associated with building binary installation packages are structured and are common across packages, it is possible to automate most tasks. Accurate automation has the benefit of increasing the uniformity of package configuration and operation across packages.

mkpkg has automated or partially automated the following tasks:
- package manifest generation
- shared library dependency detection
- fileset and subproduct generation
- assigning files to filesets
- control script generation
- error checking

Package Manifest Generation

The first task faced by most package creators is creating a manifest or list of all the files installed as part of the package. It is critical that all files be included in the package, so it is important to reduce human error. For some packages, creating a manifest is a trivial task that can be accomplished easily by visual inspection of the software. However, packages often include dozens of files, and some packages include thousands of files. In these cases, it is very difficult to manually generate a complete and accurate manifest.

mkpkg can automatically generate a package manifest that includes all files installed as part of the software and may include some files not belonging to the package.

Shared Library Dependency Detection

mkpkg automatically detects all shared library dependencies. It checks every file in the product to discover which shared libraries are used by the product. It has a list of all shared libraries on the system and the name of the fileset that contains each library. *mkpkg* then automatically marks as a co-dependency each fileset containing a shared library needed by an executable.

When I first developed *mkpkg*, the vast majority of bugs were caused by shared library dependencies that I had overlooked. Once I added this module to *mkpkg* the number of bug reports diminished dramatically.

Fileset and Subproduct Creation

Software Distributor allows a given product to contain filesets and subproducts (groups of filesets).

Hewlett-Packard has extensive standards for fileset and subproduct naming and semantics. For example, English-language manual pages should be contained in the XXX-MAN fileset, while foreign-language manual pages should have a fileset per language (e.g., XXX-SPA-I-MAN for Spanish with an ISO character set). Fortunately, it is possible to use simple regular expression patterns to recognize when particular filesets are needed. Similarly, there is an extensive set of standards for subproduct naming based on the filesets in a product (e.g., the subproduct ManualsByLanguage always includes all filesets with non-English manual pages).

mkpkg has two ordered sets of rules for determining when to create filesets and subproducts. Each rule contains a regular expression, a threshold value, and a pattern. During fileset creation, the system iterates through the rules. It first creates a list of all files that match the regular expression. If the number of files is greater than the threshold value, then a fileset is created using the pattern (if necessary), and all the matching files are assigned to the fileset. The threshold value is used because some of the conventions are of the form "if there are enough XXX files, then put them in -YYY fileset." For example, "if there are enough manual pages, put them in a -MAN fileset."

Assigning Files to Filesets

The same rules that determine when to create filesets are used to assign files to filesets. This is particularly useful for large packages for which manual assignment of files to filesets would be tedious.

mkpkg uses the same pattern matching to decide how to assign each file to a fileset. Each file is assigned to the first fileset whose pattern matches the file name. By default all files that do not match any fileset are assigned to the -RUN fileset.

Control Script Generation

One of the most difficult tasks is developing all the control scripts that customize the remote system. Fortunately, most control scripts execute a handful of common tasks, and in many cases it is possible to automatically detect the need for these tasks.

Control scripts may be used at both the product and the fileset levels. *mkpkg* currently knows how to automate ten common tasks and allows the user to specify customization actions at either the product or the fileset level. The user specifies high-level actions that *mkpkg* maps into low-level script fragments for each of the ten possible control scripts. *mkpkg* only generates control scripts when necessary; it doesn't generate empty control files.

Error Checking

In general, it is very difficult to perform error checking for binary packages. However, there are a number of common errors that can be detected. *mkpkg* flags as many errors as possible, but there is still room for "pilot error."

Each attribute of a product, subproduct, or fileset can be marked as "required." Before assembling the package, *mkpkg* can check that every required attribute has an associated value. For example, the attribute "description" is required and *mkpkg* will generate an error if this attribute has been left blank.

mkpkg can also check that hard links do not cross fileset boundaries. In other words, if two files are joined by a hard link, then they must be in the same fileset. Optionally, *mkpkg* can ensure that symbolic links do not cross fileset boundaries.

Manifest Generation

mkpkg can automatically determine the product's manifest. In practice it is very accurate, but there are some occasional errors. There are two types of errors: excluding necessary files and including unrelated files. The most common problem is including unrelated files, and I have only had one package that did not include a necessary file.

The RPM documentation [3] states:

> "RPM[1] has no way to know what binaries get installed as part of make install. There is NO way to do this. Some have suggested doing a find before and after the package install. With a multi-user system, this is unacceptable as other files may be created during a package building process that have nothing to do with the package itself."

While this comment is true, we have discovered a method which finds all files that were installed by the make install and automatically eliminates common mistakes. Human interaction provides the final check.

mkpkg creates the manifest using file timestamps to detect files that were installed by the install process included with the software source. *mkpkg* creates a new file that it will use as a timestamp, then it builds and installs the software (on the publisher's machine using the install process included with the software source). It then searches (part of) the file system for files with modification or creation times that are newer than the saved timestamp.

Since the manifest generation process uses file timestamps, it reliably detects all modified or installed files in the search region. There are two ways that *mkpkg* can miss files that should be included: directories missing from the search list, and files are that aren't installed. Sometimes, software installation tools are "too smart" and don't re-install files that have already been installed. In this case, some files will not be installed by the tool and will not appear on the manifest. I do not have a solution to this problem.

[1]Section 6.8 of [3].

More commonly, extra files will appear in the manifest because the files have been modified independent of the installation process. Usually, these files are log files for system events or daemon processes. In general, the list of such files is fairly static for any given machine, so we can create a list of all such files. *mkpkg* automatically eliminates most of those files by automatically removing "spurious" files from the manifest. *mkpkg* has a global list of "spurious" files that can be modified by publishers to match the active log files on their machines.

mkpkg has a default list of directories to search for new software. Currently this list includes paths from the Software Configuration Guide [4]. *mkpkg* will only search this part of the file system for newly installed files both to reduce the probability of independent user activity generating spurious file listings and to minimize the time required to search the file system for new files.

Control Script Generation

One of the most difficult tasks when developing installation packages is developing the customization scripts. These scripts are not interactive, may only rely on a restricted subset of system functionality, and must work correctly every time or they may leave the customer's system in an inconsistent state. The guidelines for writing control scripts are extensive and arcane.

Each control script is used during different phases of software installation, and there are many subtle issues regarding the roles of each script. In particular, there are some very subtle issues when software is installed on a disk shared by several computers, since, in this case, some scripts are executed only on the machine that copies the bits, while other scripts are executed on every machine that uses the software.

mkpkg provides a mechanism for automatically generating the control scripts for several customization actions, including PATH file updates, configuration file installation, removing obsolete files, adding a kernel driver or parameter, adding new users and groups, appending a fragment to a (configuration) file, and starting a daemon process. In addition, *mkpkg* can automatically detect when some of these actions are required and will automatically create the necessary customization actions.

Each customization task has a "work ticket" that specifies the type of task and any parameters. Each product and fileset has a list of these "work tickets" containing all its customization tasks.

Control Files

Table 2 shows the nine control files used by Software Distributor to customize software installations. An installation package may contain any or all of these scripts. Control scripts fall into three basic categories: installation, verification, and de-installation. The installation scripts preinstall, postinstall, and configure are executed when the software is installed on the computer, and they should install and configure the software so that it may be removed safely in the future. The verification scripts checkinstall, verify, and checkremove do not perform any work. The de-installation scripts preremove, postremove, and unconfigure are designed to remove most of the customizations added by the installation phase. This is an exceptionally difficult process to perfect, especially since some customizations should not be removed because the system may now depend upon them. For example, it might be a bad idea to remove a user since the customer may have created files using that user-id.

checkinstall	executed before bits are installed to check that the software can be installed; no side-effects
preinstall	executed before bits are copied to system in preparation for installation and customization, e.g., save original versions of configuration files
postinstall	executed after bits are copied to system. Completes customizations that are shared by all systems using network bits.
configure	may be executed independently and should be executed on every system using the software. Does system-specific customizations, e.g., add-a-user.
verify	verifies that the software is properly installed and configured. No side-effects.
checkremove	executed before the bits are removed. Checks that a fileset or product may be removed.
preremove	expected before the bits are deleted. Prepares the system and the software for removal.
postremove	executed after the bits are removed. Cleans up and removes any leftover mess.
unconfigure	executed after the bits are removed. Removes (some) system-specific customizations. Not all customizations should be undone.

Table 2: Nine control files for installation and customization.

Control File Generation

When *mkpkg* is creating a product or fileset, it iterates through the work tickets to create a sequence of code fragments for each control script. If a work ticket specifies a control action in that script, then it generates a code fragment by filling in a template with specifics from the work ticket. *mkpkg* creates each control script by concatenating the relevant code fragments.

Control Actions

The basic customization actions include: adding directories to the PATH files, adding new users or groups to the system, installing configuration files, inserting fragments to system files, removing obsolete files, adding a kernel driver or parameter, starting a daemon process and adding cron actions.

I have built over three hundred software installation packages for a wide variety of public domain applications, such as database systems, editors, compilers, and games. These software packages vary widely in many ways, but they have needed only a few types of customization. I believe that the entire body of software that I have managed needs only those customizations that have already been automated by *mkpkg*.

I was able to obtain a copy of all the control scripts written for all the software shipped by Hewlett-Packard for HP-UX using Software Distributor. I examined nearly all of the control scripts, and I think that most of the customization actions needed by Hewlett-Packard are already included in this list.

Custom Scripts

Since *mkpkg* only contains built-in customization detection and handling for a few common actions, *mkpkg* provides a mechanism for publishers to execute their own custom scripts. ‘‘Custom’’ scripts allow the user to specify that a given script be executed as part of a given control script phase. The given script is included in the package as a control script. *mkpkg* creates a control script fragment that executes the given script and retains the exit code.

PATH File Components

In HP-UX 10.x, there are three files /etc/PATH, /etc/MANPATH, and /etc/SHLIB_PATH that provide each user with default values for login shell environment variables. Any package that has its own directory tree would probably need to modify these files. For example, the new standard for independent software packages recommends that packages be installed under /opt/package/{bin,lib,etc,man,...}, so each package would require adding path elements to each of the path files. Fortunately, it is possible to automatically recognize that a fileset requires control script actions using filename pattern matching and file type checks.

Configuration File Installation

In HP-UX 10.x, configuration files should not be installed directly into place because end-users may modify configuration files. In addition, in a network file system environment, a package may be installed on one machine into a shared directory, and then run by each of the other machines after ‘‘configuration.’’ Consequently, the configure scripts need to copy configuration files into an unshared location. By convention, configuration files are contained under the /etc/ tree, but they may be located elsewhere. Although, the system only automatically detects configuration files in some cases, *mkpkg* users may specify additional configuration files.

Since configuration files must be installed into a staging location, *mkpkg* searches for files that are slated for installation directly into the configuration area (/etc/), changes the installation location to the staging area (/etc/newconfig/), and creates a work item. For files slated for installation in the staging area, *mkpkg* creates the work item to move the configuration file into place.

New Users and Groups

Some packages require that a particular user or group own files. If it is not in the standard set of users and groups for UNIX machines, then it must be installed on the system. For example, many database systems need to run as a specific user-id, and all of the database's files are owned by that user-id. *mkpkg* can generate the control script fragments that create and remove user-ids and group-ids. In many cases, *mkpkg* can automatically detect that software requires a new user-id or group-id by examining the ownership of all the files in the product or fileset. If the software has user or group ownership by any user-id or group-id that is not in the basic set of users and groups shipped with every system, then *mkpkg* needs to create a new user-id or group-id.

mkpkg-generated control scripts do not modify system files directly, but use the system programs useradd, groupadd, userdel, and groupdel to update the system.

System File Modification

Some packages need to modify existing system files. There is a class of system files that contains system configuration information and that is relatively insensitive to the ordering or location of entries. For example, /etc/inittab contains a list of processes that should be started or stopped on entry and exit from various run-levels. Each entry is a single line, and the lines are position-insensitive.

mkpkg generates the control scripts that can add or remove a line (or lines) to such files. *mkpkg* needs to know the file name and line (or lines) to insert. During customization, the generated control scripts check the file for the specified lines. If they are not present, then they are appended to the system file. Decustomization may need to remove these lines from the file.

Crontab Entries

The cron system is used to execute scheduled and repetitive actions. Each user may have an individual cron schedule. cron uses a structured configuration file, called a crontab, to control its actions. The standard system file modification action would be sufficient for cron, except for the fact that the crontab should not be modified directly. One gets a copy of the current crontab file by executing crontab -l and then updates the crontab by executing crontab <crontab> as the user whose cron schedule is being updated.

Starting a Daemon Process

Some software contains daemon processes. In many cases, the packages modify one or more system files so the daemons will be restarted automatically after a reboot. However, it is often useful to start these daemon processes immediately, without requiring a reboot. This task ensures that the daemon is started automatically on the end-user's machine during package customization.

Architecture

mkpkg has a very modular design that provides a framework for adding new modules and functionality. The user requests that *mkpkg* execute actions, such as "create the product manifest." Actions are composed of sequences of operations. The operation is the basic unit of functionality.

mkpkg executes each operation within an action in sequence. The operations may return an error code (in which case *mkpkg* may ask the packager if s/he would like to abort) and *mkpkg* adds text to the operation log. Operations have a uniform function interface. Operations are intended to function without user interaction, since *mkpkg* can be used via either a command-line interface or a GUI.

New functionality is added to the system by developing new operations, and then adding them to the appropriate action list or creating a new action.

Most of the internal structure of the system, its interaction, and user interface are all defined by data structures that specify how various pieces interact. In this way the basic code is often very simple and many pieces of the system can be reused easily and often. Sometimes the data structures are code fragments that get dynamically executed by the Tcl interpreter.

Data Structures

mkpkg's greatest weakness is its data structures for storing package configuration information; *mkpkg* uses Tcl arrays as the basic data structure container. There are two global arrays: database and product. database contains the system information that is used by all packages created on that system, while product contains the information relevant to a particular product.

The array index is a comma-separated list of defining attributes of the data value. The entire context for a given piece of information is encoded in the index. For example, the list of prerequisites for *mkpkg*'s fileset mkpkg-BIN is in the array element:

```
product(mkpkg,fileset,
          mkpkg-BIN,prerequisite)
```

This system is cumbersome but effective for most of *mkpkg*'s needs. As I have been developing the control script generation, its weaknesses for general hierarchical data have become more pronounced.

Operations

Operations are the basic building blocks of *mkpkg*. Each operation is atomic and may be used in many actions. Operations often modify the package or *mkpkg*'s state, but not always. For example, one action creates the timestamp file used during manifest generation, while another action builds the application. Not all actions modify *mkpkg*'s state; some are used to provide error checking.

Backends

The backends provide installation system-specific code. The backends provide two functions: *dumpPSF*, and package. *dumpPSF* creates the PSF file for the package using all the state and information available. package executes the backend-specific packaging program to create an installation package.

mkpkg is structured so that it can easily produce packages for a variety of software installation tools. In the past it has been able to create installation packages for several other installation tools.

Interfaces

There are two user interfaces for *mkpkg*: a command-line interface and a graphical user interface. The command-line interface provides access to most of the actions and functionality of *mkpkg*, but it does not have any facilities for browsing or modifying specific data fields.

The graphical interface provides access to all of the actions and functionality provided by *mkpkg*. In addition, it provides the ability to browse and edit all of the product configuration information, such as fileset manifests. Users may use the GUI to create, delete, or rename filesets and subproducts; to add or delete files from manifests; to specify customization actions; or to edit any one of the other myriad configuration items.

Advanced users may edit *mkpkg*'s data files, but this must be done with great care.

```
/usr/local/bin/less
/usr/local/bin/X11/xless
/usr/local/lib/X11/app-defaults/Xless
/usr/local/man/man1/less.1
/usr/local/man/man1/xless.1
```

Table 3: Software sources.

Developing a Package

Developing a package requires several steps; the package less will be used as an example. The first step is to prepare the software for packaging. We should be able to automatically compile and install the software correctly on our machine without human intervention. Of course, if we are building a package for pre-compiled software, we can skip compilation.

The software sources are in the directory less-1.0/, and there is a Makefile with three targets: all, install, and clean. less contains the files listed in Table 3. Since the package is small, and since it has only a few man pages, we will ship the entire product in a single fileset less-RUN.

We need to decide if we will distribute code that has been statically linked or dynamically linked. In general, software that is released as part of HP-UX will be dynamically linked, while software that is shipped by third parties or is shipped independently of HP-UX may be statically linked. The advantage of static linking is that the executables do not depend on specific shared libraries and are more likely to work correctly on a wider range of platforms, but at the cost of additional disk space consumption. Our package will be shipped with dynamically linked executables. We are now ready to begin building our Software Distributor (SD-UX) product.

Secondly, we start *mkpkg* within our project directory less-1.0, and provide *mkpkg* with enough information to be able to build, install, and locate the software in our product. The *mkpkg* interface has a menu bar across the top. Under the 'View' menu, we can see all of the 'pages' that contain information that we may need to provide, verify, or modify. Under the 'Action' menu are all the actions that we need to produce an installation package.

We are currently viewing the 'configuration' page for the product. Notice that *mkpkg* has already provided default values for many of the attributes. Some of the defaults come from the default values on the 'system' pages, but others have been computed. For example, the product name (less) and version (1.0) have been computed from the current directory name.

On the configuration page, we need to fill in the 'directory' attribute with /usr/local. This attribute is used in the PSF file, but it is also used by *mkpkg* during manifest generation to locate the files installed as part of the package. In some cases, we may not know where every file will be installed. *mkpkg* has a (long) list of directories in order to catch these wayward files.

We have told *mkpkg* where to look for installed files, now we need to tell it how to build and install our software. Go to the 'build' page under the 'View' menu and check the attributes 'build', 'install', and 'clean'. Their defaults 'make', 'make install', and 'make clean' are correct because they are the targets used by our Makefile.

We need to create the manifest, so select the menu option 'Create file list' on the 'Action' menu. This action may take a long time, since it compiles and installs the software, and then it searches your system for newly installed files. For large packages, just compiling the software may take hours, while for large systems it may take hours to just search the file system for installed files. Once this action is complete, you should see a fileset 'less-RUN' and a subproduct 'Runtime' under the 'View' menu.

You should now check every attribute on each page, correcting or providing information as necessary. You should also check that the fileset 'less-RUN' contains all the files from our package (and no more!), and that the subproduct 'Runtime' contains just one fileset. Also, 'less-RUN' should have dependencies on various filesets from OS-CORE and X11.[2]

Thirdly, we create the installation package with the 'Create dynamic package' action. This action compiles and installs the software. It then copies the software to some 'safe' place (in the parent directory of less-1.0, there should be a directory named something like less-1.0__10.20__dynamic). It then generates a PSF file (in the 'safe' place), and uses that PSF to create an installation package. The installation package is left in the parent directory.

Experiences

I started developing software installation packages in 1993 because I wanted to provide binary installation for public domain software within Hewlett-Packard. There is a network installation tool, called ninstall, which has been in widespread use within Hewlett-Packard for a long time. I wanted to build ninstall packages for common public domain software, such as emacs, so other people inside Hewlett-Packard could install the packages and not duplicate my porting, configuration, and compilation effort.

It took me a week to generate my first ninstall package, both because I had to learn how to package software and because I had to manually create the package manifest and all the PSFs. It only took a week to write the first version of *mkpkg*, which included the automatic manifest generation and PSF generation.

I used the initial version to develop binary installation packages for about 50 packages. At this point it would take me about three minutes of effort per-package to build a complete binary installation package once the software had been ported and configured. Since *mkpkg* builds the package several times during the course of the process, the actual elapsed time can be far longer. For example, TeX took a few hours while less took a few minutes.

At this point I was supporting a library of about 50 public domain software packages which could be installed by HP employees over the HP intranet. This library was very popular and I soon had thousands of internal "customers"; I also started getting bug reports. I discovered that my customers were having a lot of problems running programs that depended on a shared library that was missing on their machine. Usually the library was included in another package, but I had not marked the package dependency so the requisite libraries were not getting installed automatically. I then extended *mkpkg* to detect and manage shared library dependencies and the problems disappeared.

[2]Actually, *mkpkg* will only find these shared library dependencies if you have run the action 'Search system for shared libraries' prior to running 'Create file list'.

Using this version of *mkpkg*, I have been supporting over 250 packages. The biggest difficulty at this stage was developing customize/decustomize scripts for extraordinary packages. In addition, I found a few packages (e.g., TeX 3.1415) whose "make install" processes were so intelligent that the processes would only install certain files if they did not already exist. Since these files invariably existed on my machine, they were not installed during the "make install" phase of manifest generation and so they were not included in the manifest. There is no substitute for testing software packaging on a "clean" machine.

mkpkg was then extended so that it could generate both ninstall and update packages. This version of *mkpkg* was shared with the HP-UX Porting and Archive Centre so they could easily generate update packages of public domain software.

With the advent of HP-UX 10.0, update was replaced with SD-UX, the HP-UX version of Software Distributor, as the standard software installation tool. Since SD-UX added hierarchical structure on top of the simple fileset model used by update, *mkpkg* was rewritten to manage the product/subproduct/fileset structure. This hierarchy added a lot of complexity to *mkpkg*. For example, *mkpkg* now knows how to create filesets and subproducts based on standard naming conventions and other guidelines, and during manifest generation it automatically assigns files to the proper fileset. Internally, Hewlett-Packard has a variety of guidelines governing subproduct and fileset naming, assignment of files to filesets, and a myriad of other topics, and *mkpkg* tries to automate those guidelines whenever possible.

This hierarchical version of *mkpkg* was also shared with the HP-UX Porting and Archive Centre so they could start generating SD-UX packages. They have since used it to generate thousands of packages.

My final task was developing customize/decustomize scripts. While developing hundreds of ninstall packages I discovered that most packages require only a few, basic customization actions, so *mkpkg* was extended to automatically detect and generate customize/decustomize scripts for a variety of common actions.

Acknowledgements

I would like to thank Colin Charlton, Richard Lloyd, Rik Turnbull, and all of the dedicated people of the HP-UX Porting and Archive Centre who have put up with pre-release versions of *mkpkg* and provided valuable feedback.

I would also like to thank Shahryar Shahsavari of Hewlett-Packard Software Integration and Distribution Organization who gave me a great deal of advice and information while I was designing the automated customization script generation. I should also like to thank David Mullaney, George Williams, Mark Mayotte, Debbie Ogden and the rest of the Software Distributor team for their support and encouragement.

Finally, I would like to thank the anonymous reviewers for the useful comments, and Gretchen Phillips for her extensive feedback.

Portability

mkpkg should be portable to other operating systems with a minimum of effort. *mkpkg* was developed on HP-UX, and it uses HP-UX specific tools for certain tasks, such as determining the shared libraries used by an executable. However, *mkpkg* has been ported to three installation tools (ninstall, update, and SD-UX), so adding another back-end for a new installation tool should not be too difficult. I have been intending to port *mkpkg* to Linux/RPM for over a year, but have not yet found the time.

I anticipate that the biggest porting problems will be caused by control script detection and generation because of operating system specific conventions and tools for many system administration functions.

Conclusions

mkpkg dramatically simplifies the process of creating installation packages, by automating most parts of the software package creation process. Using *mkpkg* a skilled user can create complex binary installation packages for Software Distributor in a few minutes of effort, a process which used to take hours or days.

Binary installation packages are very useful, but they have primarily developed and distributed by large software and operating system vendors because they are so difficult to develop. By dramatically reducing the effort and complexity associated with developing binary installation packages, it should now be possible for harried system administrators and MIS support staff to develop their own binary installation packages for software that they support and redistribute with their organization.

mkpkg is available at: http://www.hpl.hp.com/personal/Carl_Staelin/mkpkg .

Bibliography

[1] *Managing HP-UX Software with SD-UX.* Hewlett-Packard. Part Number B2355-90080. 1995.

[2] Edward Bailey, *Maximum RPM: Taking the Red Hat Package Manager to the Limit*, Red Hat Software, Durham, North Carolina, 1997.

[3] *RPM HOW-TO*, http://www.rpm.org/support/RPM-HOWTO.html .

[4] *Software Distribution Standard*, The HP-UX Porting and Archive Centre, http://hpux.csc.liv.ac.uk/hppd/standard.html .

[5] *Standard for Information Technology – Portable Operating System Interface (POSIX) System Administration*. IEEE POSIX draft P1387.2/D13, April 1994.

[6] Scott Hazen Mueller, *Good programs, lousy installation*. ;login: (21)3:36-38, USENIX. June 1996.

Appendix A: Glossary

Bundle: A collection of products and filesets that are installed as a unit by Software Distributor.

Control script: A script that is contained in a product or fileset and which is used by Software Distributor to check or modify the system state during software installation or de-installation.

Customization Actions: Actions that are not standard and that occur during software installation and de-installation. *mkpkg* customization typically modifies the system configuration so that the software runs correctly. These actions occur without user interaction.

Dependencies: An attribute of a package that indicates whether the package requires another package to work properly. Dependencies may be either 'prerequisite' (if the package must be installed before the current package is installed) or 'corequisite' (if the package must be installed before the current package is executed).

Fileset: The atomic unit of installation. Contains files and customization scripts (if applicable). May also have additional requirements, such as dependencies.

Manifest The list of all files to be installed.

Product: The primary unit of software installation in Software Distributor. It contains both subproducts and filesets and may have installation dependencies and customization actions.

Product Specification File (PSF): The file that describes the entire product or bundle.

Subproduct: A collection of filesets that are installed as a unit by Software Distributor. Subproducts are contained in products, and a product can have several subproducts. Filesets may be contained in more than one subproduct.