Redesigning src2pkg, a Linux package creation toolkit

Timothy Goya

December 4, 2009

Senior Project
COMPUTER SCIENCE DEPARTMENT
California Polytechnic State University
San Luis Obispo
2009

Contents

1 Introduction **3**

2 Background **4**
2.1 GNU Coding Standards 4
2.2 CheckInstall 4
2.3 src2pkg 1.x 4

3 Description **5**
3.1 Flexibility and coupling 5
3.2 A simple example script 5
3.3 Other interesting functions 6
3.4 Modules 6
3.4.1 Build systems 6
3.4.2 Sandboxing methods 7
3.4.3 Package formats 7

4 Evaluation **7**

5 Really Cool Bash Features **8**
5.1 /dev/tcp 8
5.2 Manipulating Binary 9
5.3 Localization 9

6 Conclusions **9**

Acknowledgments

Many thanks to Dr. Phillip Nico for his saintly patience and understanding during his time as my senior project advisor. Special thanks to Gilbert Ashley, the author of src2pkg, for his approval and support throughout this redesign. Thanks to Piete Sartain for adminstrating the src2pkg wiki and Drew Ames for the writing much of the great documentation on the src2pkg wiki. I would also like to thank Dr. Alexander Dekhtyar for acting as my advisor for my first senior project idea.

Abstract

Package managers ease installation and removal of applications. However, as the name indicates, in order for a package manager to be useful, they need packages created from upstream sources to manage. This is the purpose of src2pkg, a toolkit written in Bash shell script which automates many package creation tasks. src2pkg, however, suffers from major design flaws that cripple its ability to package some exotic upstream sources. src2pkg-ng is a prototype redesign of src2pkg that eliminates these flaws. src2pkg-ng fully supports creating packages for Slackware and Slackware-compatible variants for at least 21 upstream sources with various levels of complexity. Debian package support is incomplete. src2pkg-ng cannot create RPM packages, but can extract the metadata and files.

1 Introduction

In 2007, Ian Murdock, founder of the Debian distribution, stated that package management is "the single biggest advancement Linux has brought to the Industry" [11]. However, a package management system is completely useless if there are no packages to manage (tautologically). Debian solved this issue with pure manpower. Installing almost anything is just an *apt-get install* command away. But, even with a repository as huge as Debian's, users may still want to install applications that are not in the repository. In order to retain the advantages of package management, particularly easy removal of packages, the user would have to somehow create a package from the upstream source. In order to create a valid and correct package, a user must read a large policy manual and comprehend the underlying rational behind the recommendations. Expecting users to read a policy manual is unreasonable.

src2pkg is a package creation tool for Slackware. src2pkg automates many of the tasks done in the official SlackBuild package creation scripts and the the scripts on SlackBuilds.org. However, src2pkg contains many assumptions only valid for Slackware. Also, the src2pkg codebase has accumulated cruft over time. This project is a complete redesign that extracts the special case handling information embedded in the current code, adds support for creating packages other than for Slackware, and features a more flexible scripting API.

2 Background

There exists no distribution-independent non-interactive package creation toolkit. SlackBuilds can only create Slackware packages. rpmbuild can only create RPM packages. dpkg can only create Debian packages.

Due to the highly open-source nature of Linux package management, the most current in-depth documents on the subject are entirely online. Nearly all dead-tree resources or equivalent only discuss how to use package managers, not how to create and maintain packages.

2.1 GNU Coding Standards

The GNU Coding Standards define how GNU sources are built and installed, the *./configure && make && make install* familiar to anyone who has installed from source [2]. These conventions were widely adopted by non-GNU developers so that users could easily see how to build and install. Using the autotools build system is the standard way to fulfill these standards [3]. Alternative build systems use roughly the same process of configuration, then compile, then install. For example, imake, qmake, and CMake replace the *configure* step. Jam replaces make for the *compile* and *install* steps. SCons and Waf replace all three steps, but still keep them separate.

Many distribution packaging guides depend on the DESTDIR convention, which directs *make install* to install the application to a different directory than the one specified in PREFIX [1]. PREFIX and DESTDIR differ in that DESTDIR only applies to installation, whereas PREFIX is often used to find auxiliary data files.

2.2 CheckInstall

The predecessor to src2pkg is CheckInstall. CheckInstall tracks the changes to the filesystem in the *install* step by intercepting standard C library calls using LD_PRELOAD and the InstallWatch library. The changed files are then packaged into either RPM, DEB, or Slackware TGZ format and installed onto the system so that uninstalling the package later will properly remove all the files in the package [4]. CheckInstall works well for creating minimally compliant packages so that applications installed from source can be easily removed, but is completely inappropriate to create distributable packages. The user still has to be intimately aware of the target distribution's packaging policies. Also, since CheckInstall is completely interactive, rebuilding a package consistently or tweaking the package without possibly introducing other errors is difficult.

2.3 src2pkg 1.x

src2pkg was originally created as an alternative front-end to InstallWatch. src2pkg is driven by non-interactive scripts so that package creation runs consistently on each build. src2pkg also attempts to correct many common installer errors.

The core of src2pkg is the Bash scripting interface. The process of creating a packages is split into 16 steps. These 16 steps must be done in order because otherwise critical environment variables will not be set. A few of the steps can be safely replaced with a hand-written equivalent, but the overall process is fixed due to the tight coupling of environment variables. This tight coupling formed not as an intentional design, but because src2pkg grew organically and hacked to fit the needs of unusual upstream sources. src2pkg attempts to maintain POSIX-compliance, and therefore uses deprecated (in Bash) constructs such as the old *test* command [10].

3 Description

The src2pkg redesign, called src2pkg-ng, has three major goals. The primary goal is to increase flexibility and loosen coupling between the functions. The redesign also modularized many aspects of the building process so that supporting new downloaders, version control systems, build systems, sandboxes and package formats is much easier. Also, since src2pkg already has a significant userbase, the redesign should roughly follow the process already used for the simple common case.

3.1 Flexibility and coupling

Environment variables in Bash are global and exhibit all the undesirable properties of global variables. In order to limit these effects, the new API only uses environment variables for system configuration and temporary return values. All other required state is passed as arguments to functions. This reduction of coupling through environment variables increases flexibility and for example, enables easy support for packaging multiple upstream sources into a single package. Explicitly passing necessary data into the functions also increases clarity.

3.2 A simple example script

```
#!/usr/bin/src2pkg-ng

NAME="physfs"
VERSION="2.0.0"
BUILD="1"

BRIEF="Virtual file system for games"
DESC="PhysicsFS is a library to provide abstract access to various archives.
HOMEPAGE="http://icculus.org/physfs/"

fetch_src "http://icculus.org/physfs/downloads/physfs-$VERSION.tar.gz"
SRC="$RET_SOURCE"
check_md5sum "$RET_SOURCE" "cfc53e0c193914c9c5ab522e58737373"
```

```
make_dir src
SRC_DIR="$RET_DIR"
unpack_archive "$SRC" "$SRC_DIR"
fixup_src "$SRC_DIR"

configure_src auto "$SRC_DIR"
compile_src auto "$SRC_DIR"
make_dir pkg
PKG_DIR="$RET_DIR"
safe_install auto "$SRC_DIR" "$PKG_DIR"
install_extra "$SRC_DIR" "$PKG_DIR"

fixup_pkg "$PKG_DIR"
make_pkg "$PKG_DIR" "$NAME"
```

The basic src2pkg-ng process follows three stages: source preparation, build, and package creation. The first step of source preparation is download and verification using *fetch_ src* and *check_ md5sum*. Source preparation finishes with creating the source directory, extracting sources to that directory, and correcting various issues with sources using *make_ dir*, *unpack_ archive*, and *fixup_ src*. The build stage is done using *configure_ src*, *compile_ src*, and *safe_ install*, which respectively corresponds to *./configure*, *make*, and *make install*. *install_ extras* does special installation of known files just in case it was not done by the *safe_ install*. *fixup_ pkg* and *make_ pkg* corrects common issues in packages and creates the actual package from the files collected in *safe_ install*.

3.3 Other interesting functions

fetch_ resources downloads and verifies support files like patches and user metadata overrides in a batch rather than having to individually fetch the URL and the verify MD5 checksum. *make_ snapshot* checks out files from a version control repository and archives them into a tarball.

3.4 Modules

src2pkg-ng is modularized using Bash's indirect function calls. For example, when src2pkg-ng is downloading a source from a URL, it extracts the protocol from the URL into the variable PROTOCOL, then calls the function *download_ $PROTOCOL*. If PROTOCOL is http, then the actual function used is named download_http.

3.4.1 Build systems

The build system type is passed into *configure_ src*, *compile_ src*, and *safe_ install* as the first argument. src2pkg-ng then calls *configure_ cmd_ $TYPE*, *compile_ cmd_ $TYPE*, or *install_ cmd_ $TYPE*. The "none" type uses *true* as the build command, effectively skipping the build step. The "auto" type attempts

to detect the actual build system used by checking each supported build system using the function *is_build_$TYPE* until it receives a positive answer. If no supported build system is found then it returns the command for "none".

3.4.2 Sandboxing methods

A src2pkg-ng compatible sandbox is a shell script or executable that uses a particular command-line interface and a basic model. The basic model of a src2pkg-ng sandbox is a chrooted unionfs mount with exclusions. This model is controlled using three types of parameters (-w, -r, and -e). Zero or one write layers are specified using -w. Any number of read layers are specified using -r. Directories that are accessible directly from inside the sandbox ("exclusions") are specified using -e. Sandboxes must also implement the -c parameter to do basic sanity checks, like ensuring that the user is running with the correct privileges.

A unionfs mount uses the concept of the "directory stack". Directories higher in the stack are given priority when accessing files inside the union. The directory stack can be thought of as trays of cups where each cup is a file that exists inside that directory. If two cups are in the same X-Y location (pathname), the cup in the higher tray will catch a ball dropped (accessed). Accesses to files in exclusion directories bypass the entire stack and uses the actual file in the actual root filesystem directly.

The script sandbox/basic is an implementation of a src2pkg-ng compatible sandbox using exactly the basic model. David A. Wheeler has written a fairly thorough essay on various methods to create sandboxes entitled "Automating DESTDIR for Packaging" [1].

3.4.3 Package formats

In *make_pkg*, src2pkg-ng calls the package creation handler make_pkg_$PKG_FORMAT. PKG_FORMAT is either set by the user manually, or each package module detects whether it is appropriate for the system and sets PKG_FORMAT if it is. Package modules should also set PKG_COMPRESSION if necessary. The package creation handler is responsible for generate format specific metadata from package files or format independent metadata. These metadata files are placed into metadata/ folder in the package directory for user inspection. Package creation handlers should allow user overrides for any format specific metadata.

4 Evaluation

src2pkg-ng uses "dog food" testing to determine that the API flexible enough to handle strange upstream sources. That is, actually using the tool to create packages for many different applications. This dogfooding also creates a large wealth of examples for users to inspect and modify to their needs.

Application	special (beyond simple example) src2pkg-ng features used
AbiWord	autopatching, packaging from two source tarballs
CMake	bootstrap (uses dummy sandbox)
Dillo	none
FLTK2	SVN snapshot creation
Gens	none
Glib2	none
GNUmeric	none
GOffice	semi-auto patching
IceWM	Batch resource fetching, automatic handling of xinitrc and config files
libgsf	none
libsoup	none
Lyx	none
PCManFM	none
PhysFS	none
Rarian	none
Scite	uses two build stages (scintilla library, then scite app)
SDL	multiple sources built into single package
t1utils	none
Transmission	none
unionfs-fuse	bootstrap (uses dummy sandbox)
wv	none

5 Really Cool Bash Features

Bash is a surprisingly powerful language due to many features in Bash that do not exist in the standard POSIX Bourne shell. src2pkg-ng uses many of these non-standard features to increase clarity and reduce maintenance effort.

5.1 /dev/tcp

Bash supports creating TCP socket connections by performing file redirection on the pseudo-device /dev/tcp/$HOST/$PORT, where HOST and PORT are the hostname and port number of the desired connection. For example, the current time can be acquired from nist.gov using [8]:

```
cat < /dev/tcp/time.nist.gov/13
```

A simple HTTP session could be done like:

```
exec 5<>/dev/tcp/www.google.com/80
echo -e "GET / HTTP/1.0\n" >&5
cat <&5
```

src2pkg-ng expands on this latter example to use as a fallback if no FTP or HTTP downloader (such as wget or curl) exists on the system.

5.2 Manipulating Binary

All Unix shells are good at manipulating printable strings. However, shells easily choke on NULLs in binary data. One of the most effective ways of processes binary is to hexdump the data in order to turn the binary into a nice printable string. This combination of dd, od, and tr read a number of bytes from a file at a specified offset and prints a raw unannotated hexdump:

```
dd if="$FILE" bs=1 count=$COUNT skip=$OFFSET 2> /dev/null | \
od -A n -t x1 | \
tr -d $'\n *'
```

src2pkg-ng uses this technique to read the binary header on RPM files as described by the Linux Standard Base [7].

5.3 Localization

Bash supports using *gettext* to localize scripts [9]. Strings using $"..." are marked for translation. Once all translatable strings are marked, Bash can scan the source (using the `--dump-po-strings` command line option) for the marked strings in order to generate a POT file. The POT file is then modified for a particular language to create PO files. *msgfmt* converts the text PO files to binary MO files suitable for use by the gettext library. When running the script, gettext automatically finds the appropriate translation based on locale settings and $TEXTDOMAIN environment variable.

6 Conclusions

src2pkg-ng can successfully create correct Slackware packages for many upstream sources.

Support for creating RPM packages was not implemented during the time covered by this report. Also, support for creating Debian packages is still preliminary and does not properly cover many important details. src2pkg-ng needs volunteers maintainers that are familiar with the requirements of these alternative package formats.

Other future directions include an elegant method to split development, internationalization, or other types of files into tagged packages such as -devel or -nls. src2pkg-ng would also benefit greatly from a sandbox that only requires user privileges.

References

[1] David A. Wheeler. Automating DESTDIR for Packaging. 19 Feb 2009. <http://www.dwheeler.com/essays/automating-destdir.html>

[2] Free Software Foundation, Inc. GNU Coding Standards. 29 Nov 2009. <http://www.gnu.org/prep/standards/standards.html#Managing-Releases>

[3] Free Software Foundation, Inc. Why Autotools. 29 Nov 2009. <http://www.gnu.org/software/hello/manual/automake/Why-Autotools.html>

[4] Duran, Felipe. CheckInstall README. 1 Nov 2006 <http://www.asic-linux.com.mx/~izto/checkinstall/docs/README>

[5] Jackson, Ian and Scwarz, Christian. Debian Policy Manual. 16 Aug 2009. <http://www.debian.org/doc/debian-policy/>

[6] Red Hat, Inc. and others. Fedora Package:Guidelines. 3 Nov 2009. <http://fedoraproject.org/wiki/Packaging/Guidelines>

[7] Linux Standard Base. <http://refspecs.linux-foundation.org/LSB_3.2.0/LSB-Core-generic/LSB-Core-generic/book1.html>

[8] Cooper, Mendel. Advanced Bash-Scripting Guide. 26 Jan 2009. <http://tldp.org/LDP/abs/html/>

[9] Wooledge, Greg. How to add localization support to your bash scripts. 1 Mar 2009. <http://mywiki.wooledge.org/BashFAQ/098>

[10] Wooledge, Greg and others. What is the difference between test, [and [[?. 6 Feb 2009. <http://mywiki.wooledge.org/BashFAQ/031>

[11] Murdock, Ian. How package management changed everything. 21 Jul 2007. <http://ianmurdock.com/solaris/how-package-management-changed-everything/>