# dpkg 2.0

Package Manager Design

**Prepared for:** HP

**By:** Canonical Limited

May 2, 2006

i n v e n t

# 1 Introduction

This document outlines the design for a new package manager intended to replace the `dpkg` package manager used by Debian and its derivatives, including Ubuntu.

The existing software has served Debian well for the past decade but has found to be both inflexible and to have some problems that are difficult to fix with the current implementation. This new design takes what worked from the original implementation and evolves it by drawing in new ideas.

While the title of this document refers to the new software as `dpkg 2.0` that is not intended to be anything more than a convenient moniker during development. It may be appropriate or desired for a new name to be chosen to avoid confusion, given its fundamentally different operation.

# 2 General Changes

## 2.1 Source and Binary Formats

Paradoxically for a package manager design, no specification for the format of source packages or the binary container will be made in this document.

Source packages will be managed by a different piece of software, or perhaps several competing ones. The existing source package tools and `debian` directory format can be used with only minor modifications, and will probably form the first software set.

The intent of separating the binary and source handling into different software is to encourage innovation and wider adoption of the package manager. It would be equally valid to construct packages from RPM-style `spec` files or directly from revision control systems.

The binary container format is intended to be fixed, however at this point is only considered a minor implementation detail. A winning container format would allow files and their meta-data to co-exist along with general information about the package itself. The advantages of competing systems such as `tar` vs. `cpio` have not yet been weighed, and it is not expected that they would be until relatively late in the implementation process.

Two features of the new container format are considered essential; support for integral check-sums or similar of the package file itself so that it can be verified as undamaged before unpacking and digital signatures to verify the origin.

## 2.2 Library Design

One of the fundamental changes to the design of the software is that all of the functionality will be exposed in one or more shared libraries.

It is with these libraries that supporting software such as APT or SMART is expected to interact, allowing them a greater degree of flexibility and control over the process.

The libraries themselves would be properly versioned, with a stable and well-documented API/ABI to ease the job of authors of these tools and bindings to other languages.

The provided command-line tools will be focused purely on user and system administrator interaction, and will be very thin wrappers around the libraries.

## 2.3 Atomic Operation

Atomic operation is a focus of the new design, ensuring that all changes can be rolled back if an operation fails. A journal will be used to ensure that all filesystem changes can be repaired or reverted in the event of a software crash or power loss.

This journal will only be flushed once an operation has been considered complete, and the system to be in the desired final state. This desired state can be specified by software using the libraries; APT may be content with a package being left in an *un-configured* state (see Section 5), another front-end may prefer that packages be reverted unless they are *configured* while SMART may want all packages to be reverted if one package in the set fails to be *configured* so it can reject the entire solution.

Because supporting software will be expected to use the shared library and specify its desired level of atomicity, it leaves the command-line tools to behave in a manner that most systems administrators would expect.

This may include giving the administrator the option to keep data in the journal so that entire upgrades can be reverted after completion, or so that upgrade issues can be debugged by simply asking for a copy of the journal.

A user using the command-line tools to install a package can expect that if a problem occurs the system will be left in exactly the same state as before the command is run. This includes problems such as missing dependencies which would currently leave the package unpacked but *un-configured*.

### 2.4 Focus on Installed Packages

Unlike `dpkg`, this new design is entirely focused on those packages that are installed on the system. No record of available packages is kept and when a package is entirely removed from the system all record of it is also lost.

Information about such packages is considered to be the job of front-ends like APT and SMART. The `dselect` front-end would live on as a separate piece of software using the new shared libraries to access information about installed packages, just as APT, etc. will do.

## 3 Unpacking

The core operation of any package manager is the process by which it unpacks the binary container onto the disk, either as a fresh installation or an upgrade of an existing installation.

This is an operation this design changes fundamentally from `dpkg` and several new concepts need to be introduced.

### 3.1 File Meta-data

Each file, directory or special object in the package is accompanied by meta-data describing it. This meta-data is stored on the system once the package is installed and can be used both during an upgrade of the package and for verification and repair of an installed package.

Typical data may include:

- Destination filename.
- Type of object.
- *Class* of object (see Section 3.3).
- Permissions.
- Ownership.
- Digest for verification.
- Object-specific properties, e.g. major and minor number for devices or target for symbolic links.

Additional tools and front-ends may add any additional data they wish, including fields that the package manager itself may not use or know about.

It's also permitted for meta-data to be provided for a file not shipped in the binary container, for example information about files generated by maintainer scripts or even during normal operation of the package. This can be useful to relate log files to the package, and ensure they are removed when the package is purged.

Logically this means meta-data about files can also be registered against a package outside of the ordinary unpacking process, by a maintainer script or other package. Functions within the shared library and a command-line tool will be provided to do just this.

It may be appropriate for the naming of meta-data fields to be regulated by a board such as the LSB or guided by policy.

## 3.2 Filters

The first phase of unpacking a package is to read the meta-data about its contents and allow any necessary changes to be made by the system.

Such changes can include allowing the system administrator or other packages to change the permissions or ownerships of an installed file, divert files to an alternate location or even prevent their installation entirely.

This is done by passing the meta-data through *filters*, small functions or shell scripts that can make any changes they deem appropriate. Any field in the meta-data can be changed or removed, new fields can be added, and the entire file can be removed so that it is not installed[1].

Filters implementing the behaviour of the existing `dpkg-divert` and `dpkg-statoverride` tools will be provided as shared library functions that the system can access. Additional filters may be provided by the system administrator in the same way or as shell scripts obeying a specified interface. These filters are applied to the contents of all packages, and the filter is expected to behave properly if it has no changes to make.

Packages may also provide filters that only operate on their own files, these are shipped as shell scripts much like the maintainer scripts but obeying the same interface as those the system administrator themselves would provide.

[1] Resulting in its removal if already installed

## 3.3 Classes

The second phase, once all changes to the meta-data have been made, is to unpack the files themselves from the container and place them in their final location, replacing existing older versions.

To ensure reliability, the existing file is backed up first and the new file placed alongside to allow an atomic replacement using the standard `rename` system call. In case of failure, the backup can be restored in the same way.

Directories and other objects need to be created in a similar way.

The actual method by which the destination is replaced by the new object is determined by the *class* it is in, obtained from the meta-data. Objects of a given class are handled by that class's *action*, again either functions in shared libraries or shell scripts obeying a specified interface.

Class actions are provided the meta-data of each new object, including the temporary location it has been unpacked into in the case of files, and the meta-data of any existing object at the destination location. The job of the class is to replace any existing object with the new one, including applying all permissions and correct ownership. If the replacement would not be legal, for example overwriting a file in another package, the class script should produce an error.

It's also permissible for the class action to chose not to replace the original object, though it should still apply any permission or ownership changes. It is also permissible for a class action to request that another class action be used, this is not considered to be changing the class but allowing classes to build on the functionality of others.

Classes are not permitted to disobey the meta-data, if the meta-data is incorrect or it is not possible to obey (e.g. unknown user or group), the class should produce an error.

The majority of files will be in the default class, for which the action is provided. Most other class actions will also likely request the default class action be used, before or after they've performed any checks or applied the necessary changes.

Existing configuration file behaviour will be implemented as a class for which the class action will also be provided, which would not replace the existing object if the user wishes to preserve their changes.

## 3.4 Cleaning Up

The last phase is only necessary during an upgrade and is to remove any files present in the old version but not in the new version. The process for this is the same as package removal documented in Section 4.

# 4 Removing

The counterpart to the unpacking of a package is the removal of one from the system, either in whole or in part due to objects being obsoleted.

To ensure that a removal can be reverted, the existing file is backed up first so it can be restored in case of failure.

Removal of an object is also performed according to the object's class by the class action, which is given the meta-data of the file to be removed and whether a removal or purge has been requested.

The class action should simply remove the object if it exists. It may emit or a warning or error if it doesn't exist, or cannot be removed. Meta-data for objects not removed remains on the system, so the package is still considered to be part-installed.

Because meta-data can exist for files not shipped in the actual package, there is no reason to deal with log files, etc. in maintainer scripts; they can be placed in a class that only provides a purge action. Such a class will likely be provided by default.

# 5 Configuring

Once packages are unpacked, they still need to be *configured* to be moved into the `installed` state. Like `dpkg`, configuration doesn't occur until the package's dependencies are installed; however unlike `dpkg` failure of a dependency's installation can cause the reversion of the package depending on it if the front-end desires that.

## 5.1 Maintainer Scripts

The existing maintainer scripts and their arguments used by `dpkg` are seen as adequate, though they may be extended to support new features.

Much of the package-level changes currently performed by maintainer scripts should be replaced by appropriate use of filters and classes, leaving the maintainer scripts to perform more specific operations such as stopping and starting daemons.

Due to the increased emphasis on atomicity of operations, maintainer scripts in general should be more aware that any changes they make must be accompanied by code to revert those changes if necessary.

### 5.2 Hooks

Once packages have been installed, the meta-data of everything unpacked and removed is passed to the *hooks* so that any final actions can be performed.

Hooks are again functions provided in shared libraries or shell-scripts implementing a specified interface. However unlike filters or classes, a hook provided by a package is run for every file on the system rather than just those in the package.

A typical use of a hook might be to run `ldconfig` if any changes to files matching `/usr/lib/*.so` are made.

## 6 Package Meta-Data

As well as its contents, it's generally useful to know a little bit about a package; both before and after installation. Such information usually includes the package name, version, description, maintainer, etc.

Except where touched on in this section, much of this is deemed trivial and not specified other than to say that the available information will be heavily based on that available in `dpkg` and RPM today; and perhaps it's worth working with a standards board such as the LSB to draft a standard for such information.

### 6.1 Fundamentals

The fundamental three pieces of information about all packages are their name, variant and version. The variant field is optional.

The combination of a package's name and variant uniquely identifies it. If a package is already installed with both the same name and variant, then attempting to install a new package, even if supposedly different, is considered a replacement and may result in the removal of the existing package.

If a package of the same name but different variant is installed, the new package may also be installed however it should take care through correct use of filters and classes to share files where possible with the installed package.

Packages of the same variant but different names are considered to be entirely different packages.

When considering dependencies, other variants of the package are excluded from fulfilling them as well as the package itself. So as a package can't conflict with itself, neither can it conflict with a variant of itself.

The version of a package may be used by filters, class actions and maintainer scripts to determine whether a replacement is to be considered an upgrade or downgrade.

Such a distinction may be important to replace the package correctly, however in most cases it should be avoided as it often results in a package only supporting upgrades. Most use cases that required that, e.g. removal of old configuration files or migration between object types, are solved by the correct use of classes so no longer necessary.

All installed variants of a package must be of the same version, it is not permitted to replace one variant without also replacing the others. Replacing one variant would result in that package being unpacked but not configured, and the other variants being deconfigured; which may not be permitted by the front-end, resulting in an error.

## 6.2 Architecture

Unlike in `dpkg`, the architecture of a package is not a fundamental piece of information. Instead, a package's fitness for installation is determined through the dependency mechanism.

Every system provides a dependency object for each architecture it can run binaries for, typical examples might be:

| | |
|---|---|
| **Older machine** | `i386`, `i486`. |
| **Modern laptop**: | `i386`, `i486`, `i586`, `i686`. |
| **AMD64 desktop**: | `amd64`, `i386`, `i486`, `i586`, `i686`. |
| **Enterprise server**: | `ia64`, `amd64`, `i386`, `i486`, `i586`, `i686`. |

Packages would then depend on those architectures that it includes binaries, shared libraries or other architecture-dependent data for. Those with no such files would simply declare no dependencies, and are equivalent to those packages currently in the `all` architecture.

Most packages intended for 32-bit Intel machines would likely depend on the `i386` architecture, and thus would be installable on any of the examples above. C++ binaries may also depend on the `i486` architecture so would also be installable on all of the examples above, but may not be installable on extremely old machines.

Packages intended for 64-bit AMD and Intel machines would depend on the `amd64` architecture and would therefore be only installable on the AMD64 desktop or enterprise server.

This flexibility also allows speciality software such as `mplayer` or `openssl` to be optimised for each processor variant, and versions that take advantage of

`i686` instructions to be provided in addition to unoptimised versions. It would be up to front-ends to pick the "best" version depending on system and user preference, and perhaps to offer each as a possibility.

The term *small architectures* has often been used to describe collections of such optimised packages, which may only number in the dozens or less.

This also allows *architectural dependency* objects to be provided by emulators such as `qemu` or virtual machines such as `mono` or Java, as long as the appropriate kernel extensions are used to support execution of those binaries. All machines can thus become *multi-arch* (see Appendix J) through installation of an emulator, and packages requiring a virtual machine, which may not even be available, can be separated into a different FTP space.

The architecture set would most likely be extended to include the underlying kernel, so packages would combine a dependency on `i386` with a dependency on `linux` to prevent them being installable on a GNU/FreeBSD box; at least one that doesn't support running Linux binaries through the usual emulation layer.

### 6.2.1 Architecture Strings

There are still two uses for traditional architecture strings, detailed below. For both of these uses, the strings will be chosen by comparing the list of architectural dependencies in the generated package with a standard table matching them against the strings; the string with the largest subset of those dependencies will be chosen.

The initial list of strings will be based on the current list of architecture names, including the special `all` name.

The first of the uses is in the filename of the binary containers themselves, and is necessary to distinguish them from each other in FTP archives, etc.

The second of the uses is for so-called *multi-arch* packages, (see Appendix J), those that contain shared libraries and can be installed alongside other architectures of themselves. These packages use this string in their variant field, combined with any other variant name, so that they may be installed together.

## 6.3 Dependencies

The set of basic dependency fields defined by `dpkg` is considered a reasonable starting point, with some possible refinements:

- `Breaks` may be introduced; roughly `Breaks` is to `Conflicts` as `Depends` is to `Pre-Depends`.

  When one package `Conflicts` another, that other package must be either removed or, in the case of a versioned `Conflicts`, upgraded and configured before the package can be unpacked.

  `Breaks` is less strict, the other package need only be de-configured or the new version unpacked and configured before the package can be configured. The package may be unpacked at any time.

  It's use is intended where an upgrade is required due to a behaviour change, or critical bug; leaving `Conflicts` for when the packages truly contain conflicting files.

  This massively reduces the burden on ordering of an upgrade, making them rather less brittle.

- the behaviour of `Conflicts` and `Replaces` may be adjusted in line with classes, depending on the result of implementation testing.

  An `Obsoletes` field may be introduced to indicate that a package is a true replacement for another, and replace the combination of the other two fields. Such a field would be policy-mandated to always be combined with a <= version to allow one's mind to be changed.

- `Provides` will be enhanced to indicate the variant, version and *features* provided.

  Support for virtual packages may be removed in favour of *features* (see below).

- `Enhances` may result in the maintainer scripts of that package being called whenever a listed package is configured. This would allow, for example, a spell-checker or dictionary to adjust the configuration of a word processor that it supports.

  Policy should also promote the use of this field more, for example all packages that currently install files in `/usr/share/initramfs-tools` should enhance `initramfs-tools`.

A major change to the dependency system is that if a package declares any architectural dependencies then all dependency packages must depend on a subset of those same architecture dependencies.

Fundamentally this means that an `i686` package will not have its dependencies fulfilled by `amd64` packages on the system.

Because architecture-independent packages have no architecture dependencies, and thus are a subset of any other set of dependencies, they are exempt from this rule and may freely depend on and be dependent on by other packages.

This rule may need to be broken where an architecture-dependent package includes a script written in an interpreted language, and doesn't care which architecture that interpreter be installed for. A more trivial, and common example would be a package depending on a binary it executes with an interface that doesn't require them to be the same architecture.

The dependency field syntax will permit the maintainer to release this restriction allowing a package with any architectural dependencies to fulfil the dependency, or only those with specified architecture dependencies.

**Example:**

- `python` is an architecture-independent package, so may depend on any package.
- `python` depends on `python-minimal`, another architecture-independent package.
- `python` also depends on `python2.4` which contains binaries, so has an architectural dependency on `amd64`.
- `python2.4` depends on `libc6`, and because that is not an architecture-independent package, it must be a `libc6` package with an architectural dependency on `amd64`.
- `python2.4` also depends on `python2.4-doc` which is an architecture-independent package, this is permitted.
- `python2.4` also depends on `debianutils`. Because that is not an architecture-independent package, normally it would need to be a `debianutils` package with an architectural dependency on `amd64`; as with `libc6`.

  However the maintainer has specified that any architecture will do, using a dependency clause that may look like:

  ```
  Depends: debianutils [any]
  ```

  This means that either the `amd64` or `i386` package of `debianutils` would be fine.

### 6.3.1 Features

A new addition to the dependency system is the introduction of *features*.

Features are arbitrary strings that serve alongside version numbers [2] to identify whether a package is compatible with a dependency.

[2] it's expected they'd be listed in the changelog

A package such as `debhelper` could use features to indicate which helpers are provided and which compatibility levels it supports. A depending package could then state that it depends on the `dh_installman` and `compat-5` features, rather than requiring the maintainer to look up the version those were introduced.

This greatly eases the job of derivatives which may have different release cycles, or back-port features to older versions.

Packages including shared libraries could use features to indicate the SONAMEs of libraries shipped, depending packages then only need depend on any package with that feature. This would be a great benefit to third-party vendors who wish to support as many distributions as possible without providing different packages.

# Appendices

These appendices give details on problems with dpkg and new use-cases that inspired many of the features of the new design. Each of them details how the solution is solved with this design.

## A Configuration File Diversion

One of the commonly encountered problems with dpkg is that neither dpkg-divert or dpkg-statoverride can be used reliably with configuration files.

This is because diversions and overrides were implemented directly into the code where they seemed appropriate, and because handling of ordinary files and configuration files follows different code paths.

With the new design all files flow through the same code path, the first stop of which is filters where the diversion or override is applied to the meta-data which is what the class action obeys.

Also because information about each file is managed in a meta-data directory, the new diversion tool can modify that after moving the configuration file so that an upgrade can still prompt to preserve changes.

## B Configuration File Merging

A common complaint is that when a user has changed a configuration file, no attempt is made to merge their changes into the new version and instead they are given two complete files to resolve without reference to what they changed.

This is simply because dpkg does not store the original file anywhere, so cannot offer anything more.

This is something that will be solved in the new design by storing the original file in the meta-data, so that it can be retrieved when needed by the class action for the configuration file class.

This design goes further, by modularising the handling of configuration files into a class; the behaviour of them can be customised, or custom solutions provided by using different classes.

Also the class action can be overridden by a graphical front-end to interact with the user in case of conflict without resorting to opening a terminal.

## C Migrating Classes

Another common problem with `dpkg` is that it is difficult to migrate files from being a configuration file to an ordinary file, or a file generated by the maintainer script. This often results in files being left in `/etc` which should have been removed as part of the upgrade process.

This is solved by treating all files in the same way, and only differing in the exact detail of how they reach their final resting place.

Migration from a configuration file to an ordinary file is as simple as changing its class, the file will remain registered and be handled properly.

Migrating to a file not shipped in the package is also fully supported, because the file can still be listed in the meta-data and assigned a class that cleans it up on package removal or purge.

However the principal reason for the former is to avoid configuration file prompts when combined with `debconf`. This is no longer a concern, one could ship the file in a class which writes the file in its action. A generic `debconf`-driven template engine could be written as a separate piece of standard software, or included with `debconf` itself.

## D Migrating Types

Another common problem with `dpkg` is actually caused by one of its features. A system administrator is permitted to replace a directory with a symlink to a different directory, so that they can split their filesystem across multiple physical disks. This change is honoured during an upgrade.

However this means that it is not possible for a new version of a package to replace a directory with a symlink, or vice-versa, because no record of what type the original was supposed to be.

Because this new design stores meta-data for each and every object in the package, it would be aware that the object on disk was supposed to be a directory and has been replaced with a symlink. It can then clean up both the target directory and the symlink before replacing it with the new non-directory object.

## E SELinux

One of the origins of classes was to support new security layers like NSA Security-Enhanced Linux (SELinux). SELinux policy can be set on each object unpacked

by a replacement class action.

This class action could obtain the intended policy from meta-data on each file, shipped by default in the package and augmented and overridden by a filter which reads the system policy.

Different classes could be used to categorise files.

# F Partial Installations

Another of the origins of classes was to support the partial installation of packages, without parts the user considers optional such as documentation and translations.

Different classes would be given to the various optional parts, each using the default class action so that they don't behave any differently by default.

When the package is installed, a filter can then be used to remove all files in a particular class from the package meta-data list before unpacking.

If the package were already installed with those optional parts, the files would be removed as if they simply weren't present in a new version.

Likewise if an optional part were required again, the package simply needs to be reinstalled without the filter, causing all of the documentation to be installed.

This is likely to be a sufficiently common request that the necessary filter will be provided by default and can be configured from the command-line interface. The list of classes excluded from a package would be stored by the filter in the package's meta-data.

# G Relocatable Packages

A common desire is for packages to be relocatable so that they may be installed under different locations to that originally intended. For example allowing a package to be installed to `/usr`, `/usr/local`, `/opt/vendor`, etc. This would be especially useful for third-party packages.

Obviously this is dependent on the software being built without any hard-coded paths, and still being able to find other components.

A filter can be used to relocate the package according to the user's configuration and store the relocated location in the package's meta-data so it need not be provided for every upgrade.

The package can even be moved to a different location during an upgrade, though special handling for the configuration files would be necessary to recognise them as the same file as ordinarily the filename is the distinguishing part.

## H User Package Installations

Following on from relocating packages, another desire is for packages to be installable into a user's own home directory.

This can be accomplished the same way, extending the filter to change the ownership of files to the current user. Obviously the filter would generate an error if it encountered a special node that cannot be created without `root` privileges or a file requiring setuid permissions.

A personal database would also be needed to hold both the list of packages installed into one's home directory and the meta-data of each files. This way they don't need write access to the master database, and don't interfere with packages already installed on the system.

## I Alternatives

This document has already detailed how the functionality of `dpkg-divert` and `dpkg-statoverride` would be implemented using filters, however there's an additional tool shipped with `dpkg` that has not yet been discussed.

`update-alternatives` allows the system administrator to select which of a compatible set of alternative programs is available with the expected path, maintained as a symlink that resolves to a package-specific path.

One option is simply to use the current system, which will work just fine as the symlinks are managed from the package's maintainer scripts and no modification to the package manager is necessary.

This new design offers a perhaps more elegant solution, which may be preferable.

The package would contain meta-data for the expected path which would be in a special alternative class, and give the name of the alternative group, the priority and the target path within the package itself. Slave alternatives would be specified in a similar manner, except without the priority which they inherit from the master.

The class action would manage the symlink, either leaving it alone because the destination is manually selected or of a higher priority, or updating it to point to the newly installed package.

Likewise removal of a package would change the symlink to that of the highest priority, if it points to the package being removed, whether manually selected or not.

Because this behaviour would be "built-in" to the package manager, much of the problems with maintainer scripts either failing to run the tool, or running it incorrectly would go away.

# J Multi-Arch

The origins of much of the new architecture system and changes to the dependency engine began with the discussions about supporting those modern processors that are able to run binaries intended for architectures other than their own native architecture.

On such processors, combined with the proper kernel and library support, it is possible for software from all possible architectures to be freely intermixed on the same system. The term *multi-arch* was coined to describe them.

The unique problem of a multi-arch system is ensuring that an `mplayer` binary intended for an `i386` can be installed on an `amd64` system, along with its dependent `i386` shared libraries which need to co-exist with the same `amd64` libraries needed by other applications.

The initial problem of fitness for installation of a package is covered in Section 6.2 by replacing simple architecture strings with more flexible architectural dependencies.

The problem of co-installation is covered in Section 6.1 with the introduction of package variants, including how to deal with `Conflicts` and other dependencies in such situations. The policy for naming the different variants based on their architectural dependencies in Section 6.2.1.

Dealing with dependencies is covered in Section 6.3; packages containing binaries would by default require libraries from packages with the same architectural dependencies, while remaining free to depend on those packages that are architecture-independent. The dependency field syntax also allows the package to indicate a package need not share architectural dependencies with itself, or that a package with a particular architectural dependency is required.

With these problems solved, the design is inherently open to multi-arch systems and indeed encourages all systems to be multi-arch through the introduction of small architectures.

However this still relies on packages being modified so that those with shared libraries install them to architecture-unique paths and do not contain any common files with each other.

While this is desirable in the long-run anyway, the design offers a short-term solution provided that the libraries contain no hard-coded paths.

A filter can be used to relocate the libraries from `/usr/lib` to a multi-arch compatible directory such as `/usr/lib/i386-linux-gnu`, such a filter would be applied to all packages on a multi-arch system.

Common files can also be handled by use of a custom class action that permits a destination file to exist provided it comes from a different variant of the same package. As all variants would still contain meta-data for that file, the file would not be removed until the last variant was removed.

Additional sanity checking could easily be performed on the common files, the meta-data for the new and currently installed files includes a digest of both. If the custom action was not convinced the file was definitely identical across the same version of all of its variants, it could compare that as well and error if different.

As much of the work currently done in maintainer scripts can be performed instead by the package manager, an estimated 75% of packages will no longer need such scripts. These scripts are a source of minor hurdles so would make the change much easier.