Package upgrades in FOSS distributions: details and challenges

Roberto Di Cosmo, Stefano Zacchiroli, Paulo Trezentos

Package Upgrades in FOSS Distributions: Details and Challenges *

Roberto Di Cosmo Stefano Zacchiroli
Université Paris Diderot, PPS, UMR 7126, France
{ dicosmo , zack }@pps.jussieu.fr

Paulo Trezentos
UNIDE / ISCTE, 1600-082 Lisbon, Portugal
Paulo.Trezentos@iscte.pt

Abstract

The *upgrade problems* faced by Free and Open Source Software distributions have characteristics not easily found elsewhere. We describe the structure of *packages* and their role in the *upgrade process*. We show that state of the art package managers have shortcomings inhibiting their ability to cope with frequent upgrade failures. We survey current countermeasures to such failures, argue that they are not satisfactory, and sketch alternative solutions.

Categories and Subject Descriptors D.2.9 [*SOFTWARE ENGINEERING*]: Management—Life cycle; K.6.3 [*MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS*]: Software Management—Software selection

General Terms Management, Reliability, Verification

Keywords FOSS, upgrade, packages, distribution, rollback

1. Introduction

Free and Open Source Software (*FOSS*) has attracted the attention of software engineers in the past decade [16, 14] due to its peculiarities. Among them, release management [11] is the most relevant for software upgrades: software bundles—like an operating system with a basic stack of applications—in the FOSS bazaar are made of components developed and released independently without a priori coordination or central authority able to control the involved parties [10]. The volunteer nature, the licensing terms, and the need to reuse, have produced a huge amount of components, which is unparalleled in the proprietary software world. Interactions among such components are non trivial, and this is the main reason why early approaches to software upgrades, where users had to manually download, compile, install, etc., were doomed to fail. FOSS *distributions* were therefore introduced around 15 years ago to reduce the complexity of installations and upgrades for final users. Distribution maintainers act as intermediaries between "upstream" software authors and users, by encapsulating software components within abstractions called *packages*.

Distributions have been really successful: nowadays every GNU/Linux user is running one of the hundreds of available distributions. Still, distributions have inherited some properties from the FOSS bazaar: complex inter-package dependencies and frequently available package upgrades. Sysadmins unsurprisingly perform package upgrades at least once a month [1]: *new software requirements* of users require new programs to be installed and old programs to be removed; *routine upgrades* are required regularly to address security issues, bug fixes, or to add new features; *release-wide upgrades* are less frequent (typically once or twice a year), but can have higher impact as a significant fraction of the installed packages are involved. Since a large part of FOSS software is installed from packages, the damages caused by failed upgrades are potentially higher than in proprietary systems.

This paper is structured around three claims. The first one is that *FOSS package upgrades have underestimated peculiarities*. The claim is supported by Section 2 which reviews all the actors involved in FOSS package upgrades—packages as the involved entities, the upgrade process and its decomposition in clear-cut phases, the possible failures which can occur during upgrades—and highlights their peculiarities. This paper provides a detailed description of FOSS package upgrade which, to the best of our knowledge, was missing from the literature.

The second claim is that *current rollback and snapshot techniques are not enough to cope with unpredictable package upgrade failures*. Rollback and snapshot techniques are the only countermeasures currently being proposed against upgrade failures. Exploiting filesystem-level details, snapshots can be taken before performing an upgrade, and the possibility to rollback to them can then be offered to users in case of failures. Section 3 gives an overview of mainstream snapshot and rollback solutions and argues that no such solution is satisfactory (in part because of excessive disk-space requirements induced by long-term upgrade rollbacks, and

* Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n° 214898.

HotSWUp'08, October 20, 2008, Nashville, Tennessee, USA.

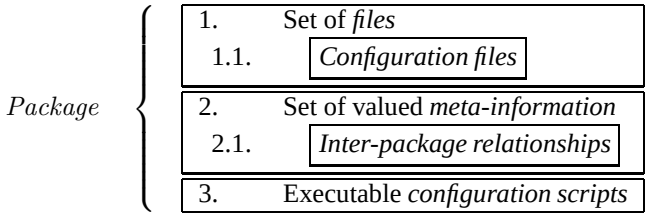
in part because it is not always possible to define the rollback scope, i.e., what should be rolled-back and what should not).

The third claim is that some of these problems can be tackled by: (1) adopting lightweight rollback techniques to address short-term rollback needs, and integrating version control system within package managers to better handle system-wide configuration files; (2) designing a domain specific language, equipped with an undo semantics, for the implementation of scripts which are executed during the upgrade process and which have side-effects outside the control of package managers. Details are given in Section 4 which also highlights a novel problem in the specification of upgrade requests, namely the need of expressing preferences to discriminate among package statuses which are equivalent from the point of view of dependency soundness, but which can be sensibly different from that of user-specified criteria.

2. Packages, upgrades, failures

Packages are abstractions defining the granularity at which users can act (add, remove, upgrade, etc.) on available software. A *distribution* is a collection of packages maintained (hopefully) coherently. The subset of a distribution corresponding to the actual packages installed on a machine is called *package status* and is meant to be altered with the *package managers* offered by a given distribution. They can be classified in two categories: *installers* which deploy individual packages on the filesystem, possibly aborting the operation if problems (e.g., unsatisfied dependencies) are encountered, and *meta-installers* which act at the inter-package level, solving dependencies and conflicts, and retrieving packages from remote repositories as needed; `dpkg` and `rpm` are representative examples of installers, `apt` and `urpmi` of meta-installers. We use the term *upgrade problem* to refer generically to any request to change the package status; such problems are usually solved by meta-installers.

Packages Abstracting over format-specific details, a *package* is a bundle of 3 main parts:



The set of files (1) is common in all software packaging solutions, it is the filesystem encoding of what the package is delivering: executable binaries, data, documentation, etc.

Configuration files (1.1) is a distinguished subset of shipped files, identifying those affecting the runtime behavior of the package and meant to be locally customized with or without package manager mediation. Configuration files need to be present in the bundle (e.g., to provide sane defaults or documentation), but need special treatment: during installation of new versions of a package, they cannot be simply overwritten, as they may contain local changes.

```
# apt-get install aterm                    1. user request
Reading package lists... Done            2. dep.resolution
Building dependency tree... Done
The following extra packages will be installed:
  libafterimage0
0 upgraded, 2 newly installed, 0 to remove and
  1786 not upgraded.
Need to get 386kB of archives.
807kB of additional disk space will be used.
Get: 1 http://ftp.debian.org libafterimage0 2.2.8-2
Get: 2 http://ftp.debian.org aterm 1.0.1-4
Fetched 386kB in 0s (410kB/s)        3. package retrieval
                                  5a. (pre-)configuration
Selecting package libafterimage0.          4. unpacking
(Reading database ... 294774 files and dirs installed.)
Unpacking libafterimage0 (libafterimage0_2.2.8-2_i386.deb)
Selecting package aterm.
Unpacking aterm (aterm_1.0.1-4_i386.deb) ...
Setting up libafterimage0 (2.2.8-2) ...
Setting up aterm (1.0.1-4) ...5b. (post-)configuration
```

Table 1. The package upgrade process

Package meta-information (2) contains information which varies from distribution to distribution. A common core provides: a unique identifier, software version, maintainer and package description, but most notably, distributions use meta-information to declare *inter-package relationships* (2.1). The relationship kinds vary with the installer, but there exists a de facto common subset including: dependencies (the need of other packages to work properly), conflicts (the inability of being co-installed with other packages), feature provisions (the ability to declare named features as provided by a given package, so that other packages can depend on them), and restricted boolean combinations of them [3].

Packages come with a set of executable configuration (or *maintainer*) scripts (3). Their purpose is to let package maintainers attach actions to hooks executed by the installer; actions usually come as POSIX shell scripts.

Three aspects of maintainer scripts are noteworthy: (a) they are ordinary programs that can do anything permitted to the installer (usually run with administrator rights); (b) the functionality of maintainer scripts can not be obtained by just shipping extra files: the scripts may customize part of the package using data which is available only in the target installation machine, and not necessarily in the package itself; sometimes the same result obtained using scripts can be precomputed (increasing package size), sometimes it can not; (c) maintainer scripts are required to work "properly": upgrade runs in which they fail trigger upgrade failures.

Upgrades Table 1 summarizes the different phases of what we call the *upgrade process*, using as an example the popular `apt` meta-installer (others follow a similar process).

Phase (1) is a user specification of how she wants the local package status to be altered. The expressiveness of the language available to formulate this *user request* varies with the meta-installer: it can be as simple as requesting the in-

stallation/removal of a single package, or as complex as `apt` pinning that allows to express preferences to discriminate among multiple versions of the same package.

An *upgrade problem* is a triple $\langle U, S_o, R\rangle$, where U is a distribution (i.e., a set of packages), $S_o \subseteq U$ is a package status, and R a user request; its *solutions* are all possible package status $S \subseteq U$, satisfying:[1]

a. The user request R is satisfied by S;
b. If S contains a package p, it contains all its dependencies;
c. S contains no two conflicting packages;
d. S has been obtained executing all required hooks and none of the involved maintainer scripts has failed.

Phase (2) performs dependency resolution: it checks whether a package status satisfying (b) and (c) exists;[2] if this is the case one is chosen in this phase.

Deploying the new status consists of package retrieval (3) and unpacking (4). Unpacking is the first phase actually changing both the package status (to keep track of installed packages) and the filesystem (to add or remove the involved files). During unpacking, configuration files are treated checking whether local configuration files have been manually modified or not; if they have, *merging* is required. The naive solution of asking the user to manually do so is still the most popular.

Intertwined with package retrieval and unpacking, there are several configuration phases (5) where maintainer scripts get executed.[3]

Failures Each phase of the upgrade process can fail. Dependency resolution can fail either because the user request is unsatisfiable (e.g., user error or inconsistent distributions [6]) or because the meta-installer is unable to find a solution. Completeness—the guarantee that a solution will be found whenever one exists—is a desirable meta-installer property [17], unfortunately missing in most meta-installers, with too few claimed exceptions [12, 19].

SAT solving has been proven to be a suitable and complete technique to solve dependencies [6], what is still missing is wide adoption. In that respect recent off-springs[4] are really promising. *Handling complex user preferences* is a novel problem for software upgrade. It boils down to letting users specify which solution to choose among all acceptable solutions. Example of preferences are policies [12, 18], like minimizing the download size or prioritizing popular packages, and also more specific requirements such as blacklisting packages maintained by an untrusted maintainer.

Package deployment can fail as well. Trivial failures, e.g., network or disk shortages, can be easily dealt with when considered in isolation: the whole upgrade process can be aborted and unpack can be undone, since all the involved files are known; no upgrade is performed so, the system is unchanged. Maintainer script failures can not be as easily undone, nor prevented. Scripts are implemented in Turing-complete languages, and all non-trivial properties about them are undecidable, including determining *a priori* their effects to be able to revert them upon failure.

A subtle type of upgrade failure deserves mention: *undetected failures*, those failures not observable by the package manager while the newly installed software can be misbehaving (e.g., a network service happily restarting after upgrade, but refusing connections). Undetected failures can take very long (weeks, months) before being discovered. Often they can be fixed by configuration tuning, but there are cases in which the desired behavior can no longer be obtained, leaving upgrade undo as the only solution (in cases where undoing the upgrade is possible).

3. Rollback & snapshot technology overview

Current countermeasures to package upgrade failures are based on the principle of undoing residual effects of failed upgrades. Three strategies have been proposed: rollbacks, filesystem snapshots, and purely functional distributions.

Rollback capabilities depend on the package manager; the most well-known implementations are: RPM transactions [13] which work at the installer level, re-creating packages as they are removed, so that they can be re-installed to undo upgrades; Apt-RPM [18] which implements transactions at the meta-installer level and additionally handles past versions of configuration files. All package-based rollback approaches can track only files which are under package manager control, and only at package manager invocation time; therefore none of such approaches can undo maintainer script effects as they can span the whole system.

Snapshots are used to cheaply save copies of physical filesystems as they were at a given time in the past. ZFS snapshot (based on *copy on write*) was the first implementation that made filesystem snapshots popular. ZFS snapshot is integrated with `apt-clone` (Nexenta OS meta-installer) to automatically take snapshots upon upgrades. The Logical Volume Manager (LVM) is a disk abstraction layer implemented by the Linux kernel, which include support for copy on write snapshots, without relying on any particular filesystem implementation.

These snapshot techniques work at the *physical* filesystem level, hence are unsuitable for recovering from upgrade failures, for various reasons. The first reason is a granularity mismatch with package managers that work at the *logi-*

[1] While (a) is installer-specific, (b) and (c) have been generalized and formalized in [6]; studies of (d) are still lacking. These are just the *functional* properties of an upgrade outcome, but there are also *non-functional* properties that can be used to choose *optimal* solutions (e.g., minimality of change, or downtime length); this issue is outside the scope of this paper. Note that while checks for (b) and (c) can be performed statically, checks for (d) can only be performed at run-time while executing scripts.

[2] The problem is at least NP-complete [3].

[3] The details depend on the available hooks; `dpkg` offers: pre/post-unpacking, pre/post-removal, and upgrade to some version [4].

[4] Apache Maven and the Eclipse P2 platform are resorting to SAT solving to manage their components and plugins, following the seminal work done by the EDOS Project (`http://www.edos-project.org`).

cal file system level: changes induced by upgrades can span several partitions and it can not be taken for granted that all support snapshots; since even the set of files of a single package can span multiple partitions, rolling back only some of them will be too prone to additional problems like "half-installed" packages. How to split the logical filesystem to support rollbacks is not clear either: while `/home` should not be rolled back (it contains user data), `/var` is a hard choice, since it contains data which are usually affected by maintainer scripts (and hence needs to be rolled back upon failure) as well as system logs and database data which usually should not be rolled back. This problem can be mitigated by a wider acceptance of the Filesystem Hierarchy Standard[5] or similar initiatives to model the purpose of specific paths.

The second reason of the unsuitability of snapshot techniques is disk usage: even though copy on write requires less space than full copying, snapshots consume as much space as the divergence between the snapshot and the live instance. The longer a snapshot is kept alive, the more physical space is needed to store deltas. Snapshots are then useful only against quickly discoverable failures (modulo the filesystem granularity problem), because it cannot be usually afforded to keep snapshots for the time span of undetected failures.

Functional distribution are embodied by NixOS [2] that proposes a functional approach to package management, where files never change after installation and are built deterministically evaluating simple functional expressions. Package deployment is based on garbage collection, hence packages can never break due to disappearing dependencies. NixOS suffers from various issues, most notably unconventional configuration handling intermixed with package building, and the fact that some actions related to upgrade deployment can not be made purely functional (e.g., user database management). NixOS made no attempt to make maintainer scripts purely functional, despite that being the place where functional purity is needed the most.

4. Towards perfected package upgrade undo

While for detectable failures trade-offs can be made using snapshot and appropriate partitioning, no fully generic solution exists to counter upgrade failures. Each of the discussed technologies focuses on one or more of the axes:

Domain : What can and should be undone upon failures (e.g., binary files, configuration files, user files)?
Time : For how long a specific upgrade can be undone?
Granularity : Does the undo of one unit imply the undo of other units? Should the unit be file, package, filesystem?

As it is unlikely that a "one size fits all" solution exists, we are pursuing[6] several research directions to improve resilience to upgrade failures in FOSS distributions:

1. Improve meta-installers by the means of (a) lightweight snapshot integration and (b) versioning;
2. Define a proper domain specific language (DSL) to be proposed as maintainer script implementation language;
3. Define ad-hoc optimized algorithms for handling complex user preferences to choose package statuses.

Simple technical improvements can sensibly improve support for upgrade failures in meta-installers. For example, porting Nexenta ideas to LVM poses no conceptual problems, and will enable GNU/Linux users to enjoy similar benefits, no matter the used filesystem. The need of LVM can be further relaxed by exploiting lightweight snapshot techniques as implemented by UnionFS [20].

Neither of these two solutions mitigates the problem of long term snapshot persistence, which will still be too expensive in terms of disk usage. Hence we also propose to exploit filesystem notifications (e.g., Linux `inotify`) to cheaply spot during package upgrades exactly which files are being modified. This would enable to trim down snapshots at the end of the upgrade, reducing space requirements.

The need of snapshots can be completely avoided by running upgrades inside controlled environments as supported by Linux out of the box (e.g., `LD_PRELOAD` to replace the system call library, and `ptrace`, a debugging interface to trace process execution). Using these approaches, one can save on the fly the files being altered by the upgrade process just before they get modified [9].

Proper handling of configuration file changes and their undo seems the easiest goal to achieve, at least at the workflow level: it is enough to properly integrate version control systems (VCSs) with meta-installers. `etckeeper`[7] is a promising example of such an approach. With `etckeeper` the whole `/etc` directory can be put under version control and enjoy integration with `apt` via hooks that commit changes to configuration files performed by upgrades, so that they can be recognized and reverted.

This does not address yet the complexity of merging user changes. A noteworthy example is the need of better integrating the merge capabilities of modern distributed VCSs. By simply keeping the pristine configuration files in a separate branch, we can isolate changes and have a clear view of the differences when manual merge is required. A related issue is the heterogeneity of languages used to write configuration files, which inhibits relying on a single diff/merge tool. To mitigate this problem we observe that for specific classes of configuration languages (e.g., XML or other structured syntaxes), syntax-level diff/merge tools can be employed, instead of the legacy VCS tools, to get rid of bogus merge failures caused by semantically irrelevant changes.

Regarding maintainer scripts, the only way we see to reliably address the undo of their effects is by properly formalizing such effects. Previous attempts to prove properties

[5] `http://www.pathname.com/fhs/`

[6] In the frame of the Mancoosi project (`http://www.mancoosi.org`)

[7] `http://joey.kitenet.net/code/etckeeper/`

about shell scripts [21, 8] have given pale results very far even from the minimal requirement of determining a priori the set of files touched by their execution, letting aside how restricted were the considered shell language subsets. Given these premises, we are skeptical that static analysis can fully solve this problem. Hence, we are developing a sound DSL equipped with undo semantics, to be proposed as the implementation language for maintainer scripts. Although it will be hard to migrate thousands of existing scripts, empirical analysis on a distribution sample has shown that most scripts are just a few lines of code, and are mostly automatically generated. The fact that scripts are maintained by distribution maintainers will enable us to test-drive the DSL on a distribution among the Mancoosi partners. The DSL will probably not be able to address all of maintainer script needs, but if it can handle most of them, we can resort to other techniques only for the remaining scripts.

As a first step in DSL design, we are applying fingerprinting techniques [15] to cluster all Debian's maintainer scripts and get a clear view of all their use cases. It is already clear that about a half of such scripts only invokes external idempotent tools to update caches of some data; this class of effects can be undone by removing the involved data—usually shipped as files by the owner package—and then re-running the script. What is still not clear is how heterogeneous are the remaining scripts which escape the former class.

Considering the intrinsic complexity of the sole dependency resolution, designing good optimizing algorithms to handle complex user preferences for package status choices is a rather ambitious goal. However, the particular shape of inter-package relationships has enabled deriving rather efficient ad-hoc dependency solvers (e.g., `edos-debcheck`). We believe similar successes can be obtained for user preferences. Hence we are not only working to apply multicriteria optimization techniques [5], but also looking for a tentative "social" solution. We are organizing a competition [7] whose participants will compete in finding the "best" algorithm to address the static part of the upgrade process. We believe the competition has chances to attract researchers attention, as it will offer real problems collected from user machines, instead of the usual *in vitro* problems.

5. Conclusion

This paper argues that upgrades in FOSS distributions have underestimated peculiarities. We have discussed the nature of packages as well as their role in the upgrade process and the potential failures. We surveyed related work and technologies, showing their shortcomings, especially in dealing with misbehaving maintainer scripts. Finally, we presented ongoing research ideas to improve the state of the art: designing a DSL for implementing maintainer scripts, and attracting the research community to work on the static part of package upgrade, including the novel problem of supporting complex user preferences among packages.

Acknowledgments The authors thank the anonymous referees for their feedback; Paulo Trezentos thanks Ines Lynce and Arlindo Oliveira for interesting discussions on this topic.

References

[1] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev.*, 41(6):221–236, 2007.

[2] E. Dolstra and A. Löh. NixOS: A purely functional Linux distribution. In *ICFP*, 2008. To appear.

[3] EDOS Project. Report on formal management of software dependencies. Deliverable D2.1 and D2.2, Mar. 2006.

[4] I. Jackson and C. Schwarz. Debian policy manual, 2008.

[5] D. Le Berre and A. Parrain. On SAT technologies for dependency management and beyond. In *ASPL* 2008.

[6] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, 199–208, Sept. 2006. IEEE CS Press.

[7] Mancoosi workpackage 1 team. Mancoosi project presentation. Deliverable D1.1, Aug. 2008.

[8] K. Mazurak and S. Zdancewic. Abash: finding bugs in bash scripts. In *PLAS '07*, 105–114, 2007. ACM.

[9] R. McQueen. Creating, reverting & manipulating filesystem changesets on Linux. Dissertation, Computer Laboratory, University of Cambridge, 2005.

[10] M. Michlmayr. Managing volunteer activity in free software projects. In *2004 USENIX, FREENIX Track*, 93–102, 2004.

[11] M. Michlmayr, F. Hunt, and D. Probert. Release management in free software projects: Practices and problems. In *Open Source Development, Adoption and Innovation*, 295–300. Springer, 2007.

[12] G. Niemeyer. Smart package manager. `http://labix.org/smart`, 2008.

[13] J. Olin Oden. Transactions and rollback with RPM. *Linux Journal*, 2004(121):1, 2004.

[14] C. Payne. On the security of open source software. *Information Systems Journal*, 12:61–78, 2002.

[15] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03*, 76–85, 2003. ACM.

[16] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 12:43–60, 2002.

[17] R. Treinen and S. Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *DebConf 8*, 2008.

[18] P. Trezentos, R. DiCosmo, S. Lauriere, M. Morgado, J. Abecasis, F. Mancinelli, and A. Oliveira. New Generation of Linux Meta-installers. *FOSDEM 2007*.

[19] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *ICSE '07*, 178–188, 2007. IEEE Computer Society.

[20] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM TOS*, 2(1):1–32, 2006.

[21] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*, 179–192. 2006.