A Model Driven Approach to Upgrade Package-Based Software Systems*

Antonio Cicchetti[1], Davide Di Ruscio[1], Patrizio Pelliccione[1], Alfonso Pierantonio[1], and Stefano Zacchiroli[2]

[1] Università degli Studi dell'Aquila, Dipartimento di Informatica, Italy
{cicchetti,diruscio,pellicci,alfonso}@di.univaq.it
[2] Université Paris Diderot, PPS, UMR 7126, France
zack@pps.jussieu.fr

Abstract. Complex software systems are often based on the abstraction of *package*, brought to popularity by Free and Open Source Software (FOSS) *distributions*. While helpful as an encapsulation layer, packages do not solve all problems of deployment, and more generally of management, of large software collections. In particular *upgrades*, which can affect several packages at once due to inter-package dependencies, often fail and do not hold good transactional properties. This paper shows how to apply *model driven* techniques to describe and manage software upgrades of FOSS distributions. It is discussed how to model static and dynamic aspects of package upgrades—the latter being the more challenging to deal with—in order to be able to *predict* common causes of upgrade failures and *undo* residual effects of failed or undesired upgrades.

Keywords: Model-driven engineering, Software change and configuration management, Metamodeling, Open source, Package.

1 Introduction

Increasingly, software systems are designed to routinely accommodate new features before and after the deployment stage. The deriving evolutionary pressure requires the system design and architecture to have enhanced quality factors: in particular, they have to retain the (user perceived as well as system-intrinsic) dependability at a satisfactory level and make component installation/removal operations less haphazard [2].

Free and Open Source Software (FOSS) distributions are among the most complex software systems known, being made of tens of thousands components evolving rapidly without centralized coordination. Similarly to other software distribution infrastructures, FOSS components are provided in "packaged" form by distribution editors. Packages define the granularity at which components are managed (installed, removed, upgraded to newer version, etc.) using *package manager* applications, such as *APT*[1] or *Apache maven*[2]. Furthermore, the system openness affords an anarchic array of dependency

* Preliminary results appeared in [1].
[1] APT howto: http://www.debian.org/doc/manuals/apt-howto
[2] Apache Maven Project: http://maven.apache.org

modalities between the adopted packages. These usually contain *maintainer scripts*, which are executed during the upgrade process to finalize component configuration. The adopted scripting languages have rarely been formally investigated, thus posing additional difficulties in understanding their side-effects which can spread throughout the system. In other words, even though a package might be viewed as a software unit, it lives without a proper component model usually defining standards (e.g., how a component interface has to be specified and how components communicate) [3,4] which facilitate integration and assure that components can be upgraded in isolation.

The problem of maintaining FOSS installations, or similarly structured software distributions, is intrinsically difficult and a satisfactory solution is missing. Today's available package managers lack several important features such as complete dependency resolution and roll-back of failed upgrades [5]. Moreover, there is no support to simulate upgrades taking the behavior of maintainer scripts into account. In fact, current tools consider only inter-package relationships which are not enough to predict side-effects and system inconsistencies encountered during upgrades.

This work is part of the MANCOOSI[3] project which aims at improving the management of complex software systems built of composable units evolving independently. In particular, this paper describes a model-driven approach to specify system configurations and available packages. Maintainer scripts are described in terms of models which abstract from the real system, but are expressive enough to predict several of their effects on package upgrades. Intuitively, we provide a more abstract interpretation of scripts, which focuses on the relevant aspects to predict the operation effects on the software distribution. To this end, models can be used to drive roll-back operations to recover previous configurations according to user decisions or after upgrade failures.

The paper is structured as follows: Section 2 describes the upgrade process of FOSS packages and, briefly, the MANCOOSI project. Section 3 describes a model driven approach to (i) specify system configurations and packages, (ii) simulate the installation of software packages, (iii) assist roll-backs. Section 4 analyzes the FOSS domain and introduces the required modeling constructs which are captured in different metamodels. Finally, Sections 5 and 6 present related and future work, respectively.

2 Packages, Upgrades and Failures

Overall, the architectures of all FOSS distributions are similar. Each user machine has a local *package status* recording which packages are currently installed and which are available from remote repositories. Package managers are used to manipulate the package status and can be classified in two categories [6]: *installers*, which deploy individual packages on the filesystem (possibly aborting the operation if problems are encountered) and *meta-installers*, which act at the inter-package level, solving dependencies and conflicts, and retrieving packages from remote repositories as needed. In an *upgrade scenario*, a user request (install, remove, upgrade to a newer version, etc.) is submitted to a meta-installer to change the system configuration status; the aim of the meta-installer is then to find a suitable *upgrade plan*, where one exists. In the rest of the

[3] MANCOOSI project: http://www.mancoosi.org

section we give a brief description of packages (as they can be found in current distributions), their role in the upgrade process, and the failures that can impact on upgrade deployment.

Packages. Abstracting over format-specific details, a *package* is a bundle of three main parts: (1) set of files, (2) meta-information, (3) maintainer scripts. The core of a package is the set of *files* (1) that ships: executable binaries, data, documentation, etc. *Configuration files* are a distinguished subset of shipped files, which are tagged as affecting the runtime behavior of the package and meant to be customized by local administrators. Configuration files need to be present in the bundle (e.g., to provide sane defaults or documentation), but need of special treatment: during upgrades they cannot be simply overwritten by newer versions, as they may contain local changes which should not be thrown away.

Package *meta-information* (2) is used by meta-installers to design upgrade plans. Details change from distribution to distribution, but a common core of meta-information consists of: a unique identifier (the name), software version, maintainer and package description, *inter-package relationships*. These relationships represent the most valuable information for dependency resolution and usually include: 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 boolean combinations of them [6].

Packages come with a set of executable *maintainer* (also known as "configuration") *scripts* (3). Their purpose is to attach actions to hooks invoked by the installer. The most common use case for maintainer scripts is to update some cache, blending together data shipped by the package, with data installed on the system, possibly by other packages. Three facets of maintainer scripts are noteworthy:
(a) maintainer scripts are full-fledged programs, written in Turing-complete programming languages. They can do anything permitted to the installer, which is usually run with system administrator rights;
(b) the functionality of maintainer scripts can not be substituted by just shipping extra files: the scripts often rely on data which are available only in the target installation machine, and not in the package itself;
(c) maintainer scripts are required to work "properly": upgrade runs, in which they fail, trigger upgrade failures and are usually detected via inspection of script exit code.

Upgrades. Table 1 summarizes the different phases of what we call the *upgrade process*, using as an example the popular `APT` meta-installer. The process starts in phase (1) with the user requesting to alter the local package status. The expressiveness of the requests varies with the meta-installer, but the aforementioned actions (install, remove, etc.) are ubiquitously supported.

Phase (2) checks whether a package satisfying dependencies and conflicts exists (the problem is at least NP-complete [6]). If this is the case one is chosen in this phase. Deploying the new status consists of package retrieval, phase (3), and unpacking, phase (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).

Table 1. The package upgrade process

```
# apt-get install libapache2-mod-php5                                     (1)
---------------------------------------------------------------------     request
Reading package lists... Done
Building dependency tree... Done
                                                                          (2)
The following NEW packages will be installed: libapache2-mod-php5        dep.
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.            resolution
Need to get 2543kB  of archives. After this operation, 5743kB of
additional disk space will be used.
---------------------------------------------------------------------
Get:1 http://va.archive.ubuntu.com hardy-updates/main libapache2-mod-php5 (3)
5.2.4-2ubuntu5.3 [2543kB]                                                 package
Fetched 2543kB in 2s(999kB)                                               retrieval
---------------------------------------------------------------------
                                                                          (5a) pre-
---------------------------------------------------------------------     configuration
Selecting package libapache2-mod-php5.
(Reading database ... 162440  files and dirs installed.)                  (4)
Unpacking libapache2-mod-php5                                             unpacking
(from .../libapache2-mod-php5_5.2.4-2ubuntu5.3_i386.deb)
---------------------------------------------------------------------     (5b) post-
Setting up libapache2-mod-php5 (5.2.4-2ubuntu5.3)                         configuration
```

Intertwined with package retrieval and unpacking, there can be several configuration phases, (exemplified by phases (5a) and (5b) in Table 1), where maintainer scripts get executed. The details depend on the available hooks; `dpkg` offers: pre/post-installation, pre/post-removal, and upgrade to some version [7].

Exemple 1. PHP5 (a scripting language integrated with the Apache web server) executes as its `postinst` (post-installation) script the following snippet, on the left hand-side:

```
#!/bin/sh
if [-e /etc/apache2/apache2.conf]; then
    a2enmod php5 >/dev/null || true
    reload_apache
fi
```

```
#!/bin/sh
if [-e /etc/apache2/apache2.conf]; then
    a2dismod php5 || true
fi
```

Note that `prerm` is executed *before* removing files from disk, which is necessary to avoid reaching an inconsistent configuration where the Apache server is configured to rely on no longer existing files. The expressiveness of inter-package dependencies is not enough to encode this kind of dependencies: Apache does not depend on `php5` (and should not, because it is useful also without it), but while `php5` is installed, Apache needs specific configuration to work in harmony with it; at the same time, such configuration would inhibit Apache to work properly once `php5` gets removed. The book-keeping of such configuration intricacies is delegated to maintainer scripts.

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 [8]) 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 unfortunately missing in most meta-installers, with too few claimed exceptions [9].

Package deployment can fail as well. Trivial failures, e.g., network or disk failures, can be easily dealt with when considered in isolation from the other upgrade phases: the whole upgrade process can be aborted and unpack can be undone, since all the involved files are known. Maintainer script failures can not be as easily undone or prevented, since all non-trivial properties about scripts are undecidable, including determining *a priori* which parts of file-system they affect to revert them *a posteriori*.

The MANCOOSI project is working to improve upgrade support in complex software systems such as FOSS distributions. On one hand the project is working on algorithms for finding optimal upgrade paths addressing complex preferences, on the other is working on models and tools to *(i)* simulate the execution of maintainer scripts, *(ii)* predict side-effects and system inconsistences which might be raised by package upgrades, and *(iii)* instruct roll-back operations to recover previous configurations according to user decisions or after upgrade failures. In the rest of this paper we introduce the approach we are using, in the context of MANCOOSI, to attack the problems of package upgrades, namely: modeling of the involved entities and upgrade simulation.

3 Proposed Approach

As discussed, the problem of maintaining FOSS installations is far from trivial and has not yet been addressed properly [5]. In particular, current package managers are neither able to *predict* nor to *counter* vast classes of upgrade failures. The reason is that package managers rely on package meta-information only (in particular on inter-package relationships), which is not expressive enough. Our proposal consists in maintaining a model-based description of the system and simulate upgrades in advance on top of it, to detect predictable upgrade failures and notify the user before the system is affected. More generally, the models are expressive enough to isolate *inconsistent configurations* (e.g., situations in which installed components rely on the presence of disappeared sub-components), which are currently not expressible as inter-package relationships.

The adoption of model-driven techniques presents several advantages: *a)* models can be given at any level of abstraction depending on the analysis and operations one would like to perform as opposed to package dependency information whose granularity is fixed and often too coarse; *b)* complex and powerful analysis techniques are already available to detect model conflicts and inconsistencies [10,11]. In particular, contradictory patterns can be specified in a structural way by referring to the domain underlying semantics in contrast with text-based tools like version control systems where conflicts are defined at a much lower level of abstraction as diverging modifications of the same lexical element.

Figure 1 depicts the proposed approach. Basically, to simulate an upgrade run, two models are taken into account: the *System Model* and the *Package Model* (see the arrow ⓐ). The former describes the state of a given system in terms of installed packages, running services, configuration files, etc. The latter provides information about the packages involved in the upgrade, in terms of inter-package relationships. Moreover, since a trustworthy simulation has to consider the behavior of the maintainer scripts which are executed during the package upgrades, the package model specifies also an abstraction of the behaviors of such scripts. There are two possible simulation outcomes: *not valid* and *valid* (see the arrows ⓒ and ⓓ, respectively). In the former case

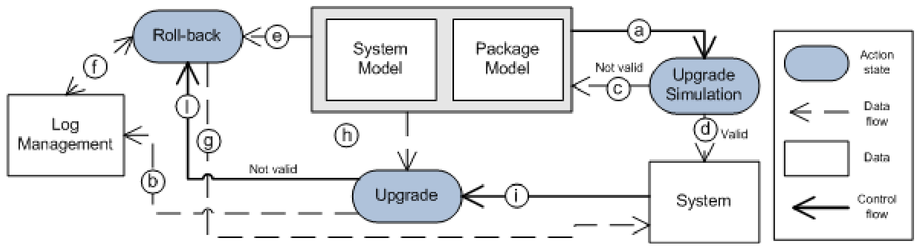

Fig. 1. Overall approach

it is granted that the upgrade on the real system will fail. Thus, before proceeding with it the problem spotted by the simulation should be fixed. In the latter case—*valid*—the upgrade on the real system can be operated (see the arrow ⓘ. However, since the models are an abstraction of the reality, upgrades failures might occur.

During package upgrades *Log models* are produced to store all the transitions between configurations (see arrow ⓑ). The information contained in the system, package, and log models (arrows ⓔ and ⓕ) are used in case of failures (arrow ⓛ) when the performed changes have to be undone to bring the system back to the previous valid configuration (arrow ⓖ). Since it is not possible to specify in detail every single part of systems and packages, trade-offs between model completeness and usefulness have been evaluated; the result of such a study has been formalized in terms of metamodels (see next section) which can be considered one of the constituting concepts of Model Driven Engineering (MDE) [12]. They are the formal definition of well-formed models, constituting the languages by which a given reality can be described in some abstract sense [13] defining an abstract interpretation of the system.

Even though the proposed approach is expressed in terms of simulations, the entailed metamodels do not mandate a simulator. Hybrid architectures composed by a package manager and metamodel implementations can be more lightweight than the simulator, yet being helpful to spot inconsistent configurations not detectable without metamodel guidance.

4 Modeling System and Packages

The simulation approach outlined in the previous section is based on a set of coordinated metamodels which have been defined by analyzing the domain of FOSS systems. In general, a metamodel specifies the modeling constructs that can be used to define models which are said to conform to a given metamodel like a program conforms to the grammar of the programming language in which it is written [13].

In this work, we have considered two complex FOSS distributions (the Debian[4] and Mandriva[5] distributions). Their analysis has induced the definition of three metamodels (see Figure 2) which describe the concepts making up a system configuration and a

[4] http://www.debian.org
[5] http://www.mandriva.com

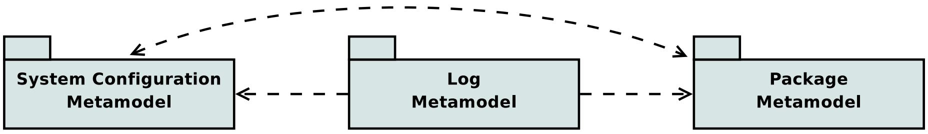

Fig. 2. Metamodels and their inter-dependencies

software package, and how to maintain the log of all upgrades. The metamodels have been defined according to an iterative process concisting of two main steps *a)* elicitation of new concepts from the domain to the metamodel *b)* validation of the formalization of the concepts by describing part of the real systems. In particular, the analysis has been performed considering the official packages released by the distributions with the aim of identifying elements that must be considered as part of the metamodels. Due to space constraints we report here only the results of the analysis, i.e., the metamodels themselves:

– the *System Configuration metamodel*, which contains all the modeling constructs necessary to make the FOSS system able to perform its intended functions. In particular it specifies installed packages, configuration files, services, filesystem state, loaded modules, shared libraries, running processes, etc. The system configuration metamodel takes into account the possible dependency between the configuration of an installed package and other package configurations. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by the proposed metamodels which embody domain concepts which are not taken into account by current package manager tools;
– the *Package metamodel*, which describes the relevant elements making up a software package. The metamodel also gives the possibility to specify the maintainer script behaviors which are currently ignored—beside mere execution—by existing package managers. In order to describe the scripts behavior, the package metamodel contains the `Statement` metaclass, see Figure. 5, that represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration;
– the *Log metamodel*, which is based on the concept of transactions that represent a set of statements that change the system configurations. Transitions can be considered as model transformations [13] which let a configuration C_1 evolve into a configuration C_2.

As depicted in Figure 2, *System Configuration* and *Package* metamodels have mutual dependencies, whereas the *Log* metamodel has only direct relations with both *System Configuration* and *Package* metamodels.

4.1 Modeling Maintainer Scripts

The most challenging part of the conducted modeling process has been modeling maintainer scripts. The reason is twofold, on one hand maintainer scripts are the entities which contribute the most to difficulties in dealing with upgrade failures; on the other hand maintainer scripts are also particularly challenging to model, due to their implementation language. Here we briefly report about the analysis (further details can be found in [14,15]) which has enabled to discover recurrent patterns in the huge amount of scripts to consider (e.g., about 25 000 on Debian Lenny). We tried to collect scripts in clusters to be able to concentrate the analysis on representatives of the equivalence classes identified. The adopted procedure for clustering has been as follows.

1. *Collect all maintainer scripts* of a given distribution. By not choosing a random subset we are sure to have collected the most representative set of scripts.
2. *Identify scripts generated from helper tools.* A large number of scripts or part of them is generated by means of “helper” tools that provide a collection of small, simple and easily understood tools that are used to automate various common aspects of building a package. Since (part of) maintainer scripts are automatically generated using these helpers and their boiler plates, we can concentrate the analysis on the helpers themselves, rather than on the result of their usage.
3. *Ignore inert script parts.* As all scripting languages, shell scripts contain parts which do not affect their computational state such as blank lines of comments. Intertwined with the removal of generated parts (to be analyzed later on) we have systematically ignored inert script parts, possibly leading upon removal to empty scripts that have been therefore ignored as a whole.
4. *Study of scripts written “by hand”.* The remaining scripts need to be more carefully studied, as they have been written from scratch by package maintainers to address a specific need, most likely not covered by any helper tool. Actually we worked on identifying recurrent templates that maintainers use when writing the scripts.

In the rest of the section, the conceived metamodels are described in more details and some explanatory models conforming to them are also provided.

4.2 Configuration Metamodel

A system configuration is the composition of artifacts necessary to make computer systems perform their intended functions [16]. In this respect, the metamodel depicted in Figure 3 specifies the main concepts which make up the configuration of a FOSS system. In particular, the `Environment` metaclass enables the specification of loaded modules, shared libraries, and running process as in the sample configuration reported in Figure 4. In such a model the reported environment is composed of the services www, and `sendmail` (see the instances `s1` and `s2`) corresponding to the running web and mail servers, respectively.

All the services provided by a system can be used once the corresponding packages have been installed (see the association between the `Configuration` and `Package` metaclasses in Figure 3) and properly configured (`PackageSetting`). Moreover, the configuration of an installed package might depend on other package configurations. For example, considering the PHP5 upgrade described in Section 2, the instances `ps1` and `ps2` of the `PackageSetting` metaclass in Figure 4 represents the settings of the installed packages `apache2`, and `libapache-mod-php5`, respectively. The former depends on the latter (see the value of the attribute `depends` of `ps1` in Figure 4) and both are also associated with the corresponding files which store their configurations.

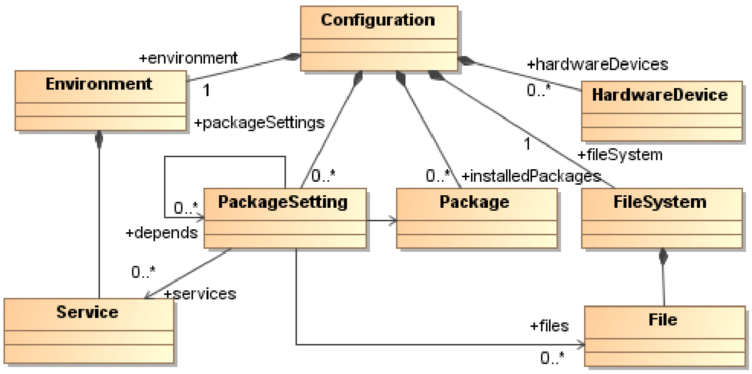

Fig. 3. Fragment of the Configuration metamodel

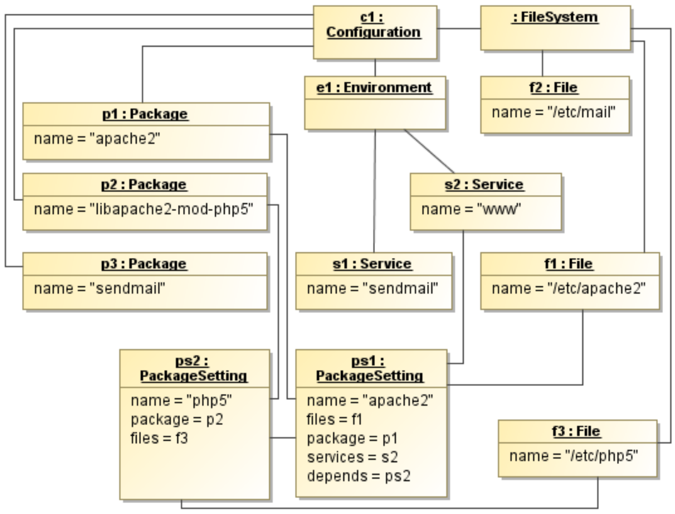

Fig. 4. Sample Configuration model

Note that at the level of package meta-information such a dependency should not be expressed, in spite of actually occurring on real systems. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by metamodeling.

The configuration metamodel gives also the possibility to specify the hardware devices of a system by means of the `HardwareDevice` metaclass. Due to space constraint the usage of such a metaclass is omitted; for more information the interested reader can found the complete metamodels on the Web.[6]

[6] http://www.di.univaq.it/diruscio/mancoosi

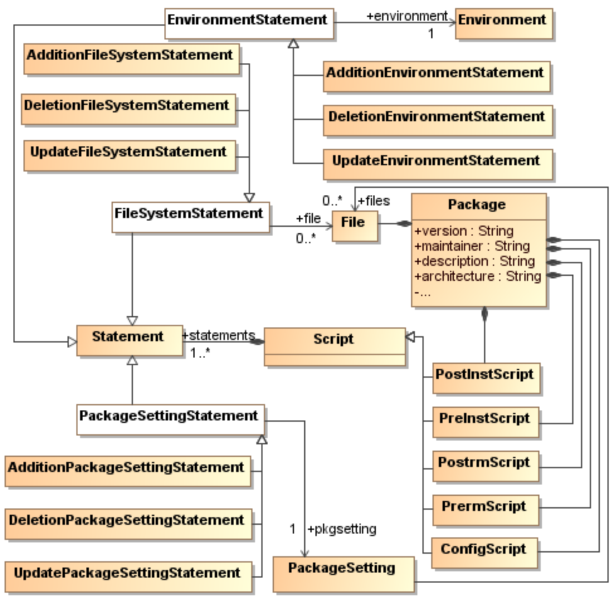

Fig. 5. Fragment of the Package metamodel

The packages which are installed on a given system are specified by means of the modeling constructs provided by the *Package metamodel* described in the next section.

4.3 Package Metamodel

The metamodel reported in Figure 5 plays a key role in the overall simulation. In fact, in addition to the information already available in current package descriptions, the concepts captured by the metamodel enable the specification of the behavior of maintainer scripts. In this respect, the metaclass `Statement` in Figure 5 represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration (`EnvironmentStatement`, `FileSystemStatement`, and `PackageSettingStatement`, respectively). For instance, the sample package model in Figure 6 reports the scripts contained in the package `libapache-mod-php5` introduced in Section 2. Due to space constraints, Figure 6 contains only the relevant elements of the *postinst* and *prerm* scripts which are represented by the elements `pis1` and `prs1`, respectively.

According to the model in Figure 6 the represented scripts update the configuration of the package `apache2` (see the element `ps1`) which depends on `libapache-mod-php5`. In particular, the element `upss2` corresponds to the statement `a2dismod` which disables the PHP5 module in the Apache configuration before removing the package `libapache-mod-php5` from the filesystem. This statement is necessary, otherwise inconsistent configurations can be reached like the one shown in Figure 7. The figure reports the sample `Configuration2` which has been reached by

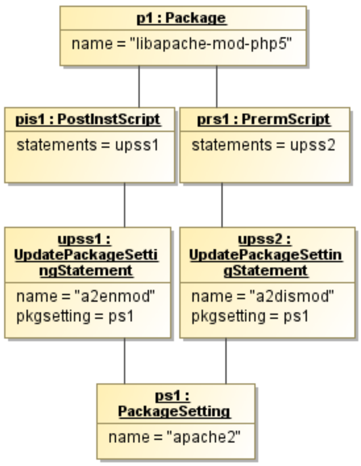

Fig. 6. Sample Package model

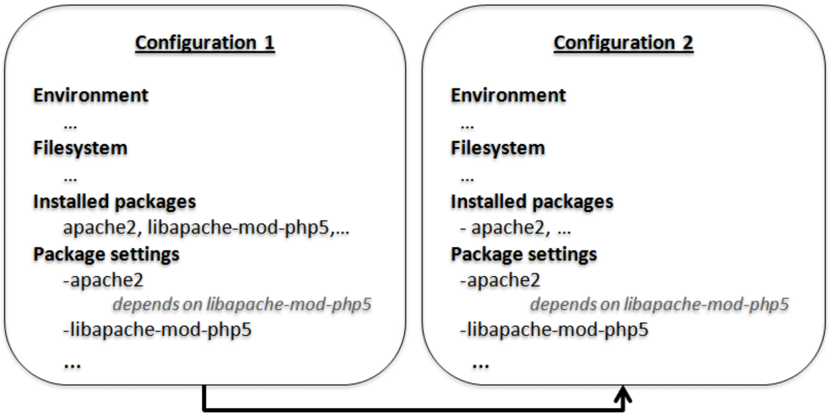

Fig. 7. Incorrect package removal

removing `libapache-mod-php5` without changing the configuration of `apache2`. Such a configuration is broken since it contains a dependency between the `apache2` and `libapache-mod-php5` package settings, whereas only the package `apache2` is installed.

Currently, the package managers are not able to predict inconsistencies like the one in Figure 7 since they take into account only information about package dependencies and conflicts. The metamodel reported in Figure 5 gives the possibility to specify an abstraction of the involved maintainer scripts which are executed during the package upgrades. This way, consistence checking possibilities are increased and trustworthy simulations of package upgrades can be operated.

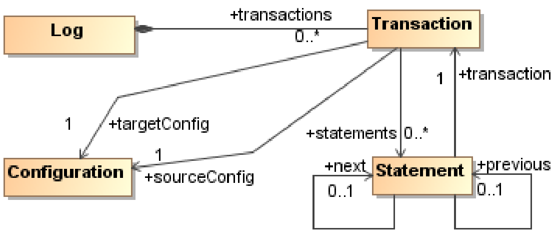

Fig. 8. Fragment of the Log metamodel

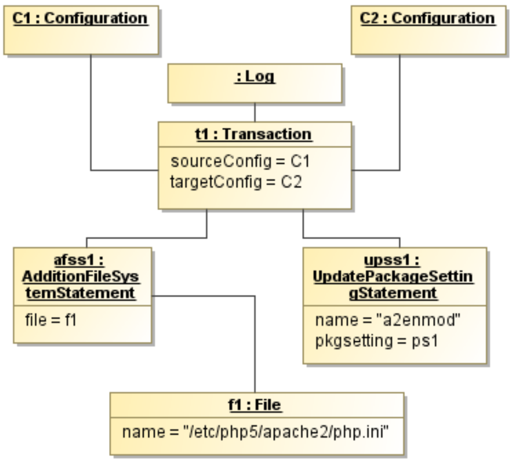

Fig. 9. Sample Log model

4.4 Log Metamodel

The metamodel depicted in Figure 8 is a step towards the development of a transactional model of upgradeability that will allow us to roll-back long upgrade history, restoring previous configurations. In particular, the metaclass `Transaction` in Figure 8 refers to the set of statements which have been executed from a source configuration leading to a target one. For instance, according to the sample log model in Figure 9, the installation of the package `libapache-mod-php5` modifies the file system (see the statement `afss1` which represents the addition of the file `f1`) and updates the Apache configuration (see the element `upss1`).

The usefulness of log models like the one in Figure 9 is manyfold and accounts for several roll-back needs:

(a) *Preference roll-back:* the user wants to recover a previous configuration, for whatever reason. For instance, the user is not in need of PHP5 anymore and wants to remove the installed package `libapache-mod-php5`. In this case, the configuration `C1` can be recovered by executing the dual operation of each statement in the transaction

between `C1` and `C2`. Note that the log models have all the information necessary to roll-back to any previous valid configuration not necessary a contiguous one;
(b) *Compensate model incompleteness*: as already discussed, upgrade simulation is not complete with respect to upgrades, and undetected failures can be encountered while deploying upgrades on the real system. For instance, the addition of the file `php.ini` during the installation of the package `libapache-mod-php5` can raise faults because of disk errors. In this case we can exploit the information stored in the log model to retrieve the fallacious statements and to roll-back to the configuration from which the broken transaction has started.
(c) *"Live" failures:* the proposed approach does not mandate to pre-simulate upgrades. In fact, it is possible as well to avoid simulation and have metamodeling supervise upgrades to detect invalid configurations as soon as they are reached. At that point, if any, log models comes into play and enable rolling back deployed changes to bring the system back to a previous valid configuration.

5 Related Works

The main difficulties related to the management of software packages depend on the existence of maintainer scripts which can have system-wide effects, and hence can not be narrowed to the involved packages only. In this respect, proposals like [17,18] represent a first step toward roll-back management. In fact, they support the re-creation of removed packages on-the-fly, so that they can be re-installed to undo an upgrade. However, such approaches can track only files which are under package manager control. Therefore, differently from us, none of such approaches can undo maintainer script side effects.

An interesting proposal to support the upgrade of a system, called NixOS, is presented in [19]. It is a purely functional distribution meaning that all static parts of a system (such as software packages, configuration files and system startup scripts) are built by pure functions. Among the main limitations of NixOS there is the fact that some actions related to upgrade deployment can not be made purely functional (e.g., user database management). Moreover, since NixOS implements a sort of "package garbage collection" (package versions are not removed as long as some other package need them) security upgrades get intrinsically more difficult due to the need of finding all versions that need upgrades. [20] proposes an attempt to monitor the upgrade process with the aim to discover what is actually being touched by an upgrade. Unfortunately, it is not sufficient to know which files have been involved in the maintainer scripts execution but we have also to consider system configuration, running services etc., as taken into account by our metamodels. Even focusing only on touched files, it is not always possible to undo an upgrade by simply recopying the old file[7]. Finally, this work can be related with techniques for static analysis of (shell) scripts. Some previous work [21] deals with SQL injection detection for PHP scripts, but it did not consider the most dynamic parts of the PHP language, quite common in scripting languages. Whereas, [22] presents an "arity" bug detection in shell scripts, but once more only considers a tiny fragment of the shell language. Both works hence are far even from the minimal

[7] This argument goes far beyond the scope of this work. For more information see [5].

requirement of determining a priori the set of files touched by script execution, letting aside how restricted were the considered shell language subsets. Given these premises, we are skeptical that static analysis can fully solve the problem illustrated in our work.

6 Conclusions and Future Works

In this paper we presented a model-driven approach to manage the upgrade of FOSS distributions and similarly structured complex, package based software systems. This approach represents an important advance with respect to the state of the art in the following directions: it provides the base on which developing features to (*i*) complete resolve packages dependencies, (*ii*) support the roll-back of failed or unwanted upgrades, and (*iii*) simulate the execution of maintainer scripts that we described in terms of models. A running example showed how the proposed models allow a reasonable description of the state of the system and representation of its evolution over time.

As future work we plan to implement these results and to develop a transactional update engine in the real context of Debian and Mandriva distributions. Moreover, the metamodels proposed in this paper will be the foundation to define a new Domain Specific Language (DSL) for maintainer script specifications.

Acknowledgements. This work is partially supported by they European Community's 7th Framework Programme (FP7/2007–2013), http://www.mancoosi.org MANCOOSI project, grant agreement n°214898.

References

1. Cicchetti, A., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards a model driven approach to upgrade complex software systems. In: Proceedings of ENASE (2009)
2. Spinellis, D., Szyperski, C.: How is open source affecting software development. IEEE Computer 21(1), 28–33 (2004)
3. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. Addison-Wesley, Reading (1998)
4. Szyperski, C.: Component technology: what, where, and how? In: Proceedings of ICSE 2003. ACM, New York (2003)
5. Di Cosmo, R., Zacchiroli, S., Trezentos, P.: Package upgrades in FOSS distributions: details and challenges. In: HotSWUp 2008, pp. 1–5. ACM, New York (2008)
6. EDOS Project: Report on formal management of software dependencies. EDOS Project Deliverable D2.1 and D2.2 (March 2006)
7. Jackson, I., Schwarz, C.: Debian policy manual (2008), `http://www.debian.org/doc/debian-policy/`
8. Mancinelli, F., Boender, J., Cosmo, R.D., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: ASE 2006, Tokyo, Japan, September 2006, pp. 199–208. IEEE CS Press, Los Alamitos (2006)
9. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Opium: Optimal package install/uninstall manager. In: ICSE 2007, pp. 178–188. IEEE Computer Society, Los Alamitos (2007)
10. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)

11. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing model conflicts in distributed development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
12. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)
13. Bézivin, J.: On the Unification Power of Models. SOSYM 4(2), 171–188 (2005)
14. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards maintainer script modernization in foss distributions. In: IWOCE 2009, pp. 11–20. ACM, New York (2009)
15. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Metamodel for describing system structure and state. Mancoosi Project deliverable D2.1 (January 2009), `http://www.mancoosi.org/deliverables/d2.1.pdf`
16. Dolstra, E., Hemel, A.: Purely functional system configuration management. In: USENIX 2007, San Diego, CA, pp. 1–6 (2007)
17. Olin Oden, J.: Transactions and rollback with rpm. Linux Journal 121, 1 (2004)
18. Trezentos, P., Di Cosmo, R., Lauriere, S., Morgado, M., Abecasis, J., Mancinelli, F., Oliveira, A.: New Generation of Linux Meta-installers. In: Research Track of FOSDEM (2007)
19. Dolstra, E., Löh, A.: NixOS: A purely functional linux distribution. In: ICFP (2008) (to appear)
20. McQueen, R.: Creating, reverting & manipulating filesystem changesets on Linux. Part II Dissertation, Computer Laboratory, University of Cambridge (May 2005)
21. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX-SS 2006, pp. 179–192 (2006)
22. Mazurak, K., Zdancewic, S.: Abash: finding bugs in bash scripts. In: PLAS 2007, pp. 105–114. ACM, New York (2007)