Supporting Software Evolution in Component-Based FOSS Systems[☆]

Roberto Di Cosmo[b], Davide Di Ruscio[a], Patrizio Pelliccione[a], Alfonso Pierantonio[a], Stefano Zacchiroli[b]

[a] *Università degli Studi dell'Aquila, Dipartimento di Informatica,*
{davide.diruscio, patrizio.pelliccione, alfonso.pierantonio}@univaq.it
[b] *Université Paris Diderot, PPS, UMR 7126, France,*
roberto@dicosmo.org, zack@pps.jussieu.fr

Abstract

FOSS (Free and Open Source Software) systems present interesting challenges in system evolution. On one hand, most FOSS systems are based on very fine-grained units of software deployment—called *packages*—which promote system evolution; on the other hand, FOSS systems are among the largest software systems known and require sophisticated static and dynamic conditions to be verified, in order to successfully deploy upgrades on user machines. The slightest error in one of these conditions can turn a routine upgrade into a system administrator nightmare.

In this paper we introduce a model-based approach to support the upgrade of FOSS systems. The approach promotes the simulation of upgrades to predict failures before affecting the real system. Both fine-grained static aspects (e.g. configuration incoherences) and dynamic aspects (e.g. the execution of configuration scripts) are taken into account, improving over the state of the art of upgrade planners. The effectiveness of the approach is validated by instantiating the approach to widely-used FOSS distributions.

1. Introduction

Any software system must undergo continuing evolution [24]. The fact that system implementations include a significant number of components does not affect this fundamental truth [25]. Software development, based on the combination of existing components, was discussed in the 1960s [31], but its practical application has not been widely explored in practice [3, 18] until recently. The management of evolution in component-based systems cannot neglect the numerous relationships, implicit or explicit, among components which might be affected when performing routine component management operations; doing so can easily lead to unusable and corrupted systems. The increasing adoption of Free and Open Source Software (FOSS) has worsened these problems [26] mainly because of a non-centralized and controlled development of system components

[☆]This work is partly supported by they European Community's 7th Framework Programme (FP7/2007–2013), Mancoosi project, grant agreement n. 214898.

which are frequently released [33]. Therefore, the ability to analyze and predict component behavior during their upgrades, e.g. installation and removal, in FOSS systems is intrinsically difficult and requires techniques, algorithms, and methods which are both expressive and computationally feasible in order to be used in practice. In particular, focusing on Linux distributions, current upgrade management tools, called package managers, are only aware of *some* static aspects of packages that can influence upgrades. At the same time package managers completely ignore relevant dynamic aspects, such as potential failures of configuration scripts that are executed during upgrade deployment. Thus, it is not surprising that an apparently innocuous package upgrade can end up with a *broken* system state [9]. Despite the relevant body of research on software evolution (see e.g. [29]), there is still the need for better predictive models which are able to anticipate the consequences of applying specific system changes [30].

In this paper, we propose an approach based on *model-driven* techniques, called Evoss (EVolution of free and Open Source Software), to enhance the prediction of upgrades in FOSS distributions. In order to make upgrade prediction more *accurate*, Evoss considers both static and dynamic aspects of upgrades. Static aspects have been modeled by enhancing the expressiveness of the representations with respect to the state of the art of package managers, enabling the detection of a larger number of undesirable configurations, such as the breakage of fine-grained dependencies among packages, currently neglected by package managers. The main dynamic aspects considered are those related to the behavior of package configuration scripts which are executed during upgrade deployment. The scripting languages in which such scripts are written have rarely been formally investigated, thus posing additional difficulties in understanding their side-effects which are pervasive and are currently ignored when planning upgrades. A notion of *simulation* is given and shown to be realizable by means of a domain-specific language (DSL) that specifies maintainer script behavior. The language includes a set of high level clauses with a well defined *transformational* semantics expressed in terms of system state modifications: each system state is given as a model and the script behavior is represented by corresponding model transformations. The proposed semantics is designed to better understand how the system evolves in a new configuration by simulating system upgrades: this allows system users to discover upgrade failures due to fallacious maintainer scripts before deploying the upgrade on the real system. In order to apply the proposed simulation approach we provide *model injectors* to automatically extract system configuration and package models from existing artifacts.

To sum up, Evoss represents an advancement, with respect to the state of the art of package managers, in the following aspects: (i) it provides a homogeneous representation of the whole system configuration in terms of models, including relevant system elements that are currently not explicitly represented, (ii) it supports the upgrade simulation with the aim to discover failures before they can affect the real system, (iii) it proposes a failure detector module able to discover problems on the configuration reached by the simulation. Examples of these problems are implicit dependencies, missing configuration files, dangling mime-type handlers, missing services, and so on.

The paper is structured as follows: Section 2 gives an overview of FOSS concepts and highlights limitations of current package management solutions. Section 3 introduces Evoss and details the modeling of the static parts of the upgrade process. Section 4 is devoted to the dynamic part and presents the DSL describing its syntax and semantics.

The validation of the proposed approach is reported in Section 5 by showing how EVOSS is able to solve some of the limitations highlighted in Section 2. Section 6 discusses some aspects related to the applicability and the full acceptance of EVOSS in real scenarios. Related work is discussed in Section 7, just before concluding with future research directions in Section 8.

2. FOSS distributions

Widely used FOSS distributions, like Debian, Ubuntu, Fedora, and Suse are based on the central notion of software *package*. Packages are assembled to build a specific software system. The recommended way of evolving such systems is to use *package manager* tools to perform system modifications by adding, removing, or replacing packages. These operations are generically called *upgrades*. Existent package managers integrate some preliminary checks, albeit extremely weak: many unpredicted upgrade failures can happen. To better understand limitations of existent package managers, Section 2.1 provides an overview of the typical elements composing the packages of FOSS distributions, and in Section 2.2 we provide a classification of possible upgrade failures, discussing their origins.

2.1. Packages and upgrades

In FOSS distributions, a *package* is a software unit $u = \langle c, d \rangle$ containing the software component c and a description d of it, also known as *metadata*. More precisely, the structure of a package $u = \langle c, d \rangle$ [12] is shown in Figure 1.

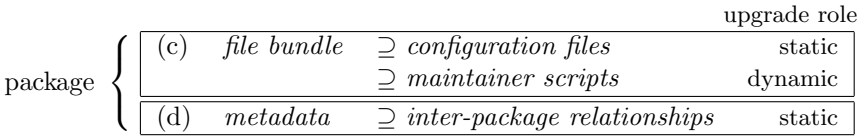


Figure 1: Structure of a package

The core of each package is a *file bundle* encoding the shipped component: executable binaries, data, documentation, etc. A distinguished subset of those files consists of *configuration files*, which affect the runtime behavior of the component and are meant to be customized. During upgrade deployment, most files play a "static" role, in the sense that they are simply copied over.

Packages also contain a set of executable *maintainer scripts*, used by package maintainers to hook custom actions into the upgrade process. Several aspects of maintainer scripts are noteworthy: (i) they play a dynamic role as they are executed *during* upgrades; (ii) they are full-fledged programs, usually written in POSIX shell language; (iii) they are run with sysadm rights and then they may perform arbitrary changes to the whole system; (iv) they cannot be replaced by just shipping extra files: they might need to access data which is available in the target installation machine, but not in the package itself; (v) they are expected to complete without errors: their failures, usually signalled by non-0 exit codes, automatically trigger upgrade failures.

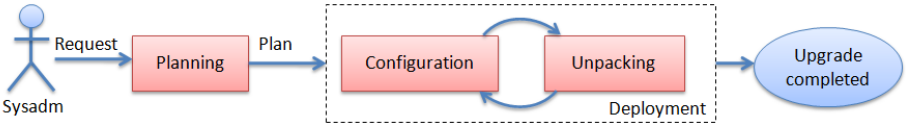


Figure 2: The upgrade process: main phases

Metadata describe package aspects needed for upgrade planning. Common metadata contain the package identifier, version, maintainer, and description. Most notably, metadata are also used to declare *inter-package relationships* such as: dependencies (the need of other packages to work properly), conflicts (the incompatibilities with other packages), and feature provisions (an indirection layer over dependencies) [26]. This information is taken into account by the package manager when performing the system upgrade.

A system configuration can be very complex: it is composed of the file system, running services and processes, environment data, memory content, etc. In practice, the most relevant part of the system state for upgrades is the *package status*, recording which packages are currently installed. Changes to the package status are performed by *package managers* that are usually separated into two types: low-level *installers*—which deploy individual packages on the system, possibly aborting the operation if problems are encountered at deploy-time—and high-level *meta-installers*—which first plan upgrades and then drive installers.

Figure 2 gives an overview of the overall *upgrade process*. The first phase—*planning*—tries to find an upgrade plan that satisfies the user request. Planning poses challenging algorithmic problems such as solving inter-package relationships (a NP-complete problem [26]) and choosing the best configuration according to user preferences [35]. Nowadays, planning is entirely delegated to meta-installers. In a typical upgrade scenario, the system administrator (or *sysadm*) requests a status change and the meta-installer devises a corresponding *upgrade plan* to be deployed by the installer. The actual *deployment* phase alternates between *unpacking* and *configuration* stages: during the former, static package files get installed on disk; during the latter, maintainer scripts are executed at the appropriate hook points. Hook points encompass pre-/post-installation hooks (i.e., before/after package unpacking), pre-/post-removal hooks, etc.

In order to figure out the role of maintainer scripts during an upgrade, we report a simple example of such scripts (see Example 1).

Example 1. *A recent Debian* `php5` *package—which ships a web scripting language—contains a post-installation (*`postinst`*, on the left) and a pre-removal script (*`prerm`*, on the right):*

```
1 #!/bin/sh
2 if [-e /etc/apache2/apache2.conf];
    ↪then
3     a2enmod php5 || true
4     reload_apache
5 fi
```

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

`postinst` *gets executed after unpacking and, in particular, after the* `Apache` *module* `php5` *files have been installed: it first takes care of enabling the module by invoking the* `a2enmod` *command on line 3, then it reloads the* `Apache` *service (line 4) to activate it. Upon* `php5` *removal, this module is disabled by invoking* `a2dismod`*.*

2.2. Upgrade failures: reasons and discussion

Current tools are able to predict a very limited set of upgrade failures before deployment: most notably, meta-installers can only detect broken dependencies by inspecting

the metadata, *before* calling the low-level installer, which would otherwise complain about them during deployment. A meta-installer knows how to check a precondition that ensures that no broken dependencies exist in the target configuration. If the precondition is not verified, no attempt will be made to deploy the (broken) upgrade plan.

Unfortunately, when trying to predict upgrade failures, existing tools only consider static package metadata. In this way they do not take into account implicit dependencies among packages that occur, for instance, because of their configuration files. For example, the package `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 bookkeeping of such configuration intricacies is delegated to the maintainer scripts shown in Example 1.

Moreover, the behaviour of the maintainer scripts is completely ignored. This leaves a wide range of failures unpredicted, some of which can be captured by using our model-driven approach.

Upgrade deployment can fail for several reasons [12]. Both experience and previous research show that failures are not hypothetical, but rather the reality of sysadm life [9]. Upgrade failures can be classified according to *when* a failure is detected: at deploy-time (usually by the installer) or later on (usually by the user). Deploy-time failures can be refined according to the specific upgrade phase in which they are detected and to whether they concern the static or dynamic part of a package.

Static deploy-time failures occur when a static requirement is violated during the upgrade: typical examples are file conflicts (two packages attempt to install the same file, violating an explicit installation policy) and dependency errors (an attempt is made to install a package violating package dependencies). In both cases, the low-level package manager fails at deploy-time, aborting the upgrade process.

Dynamic deploy-time failures occur when a maintainer script fails. They are tricky to deal with, given that shell script failures can originate from a wide range of errors, ranging from syntax errors to failures in the invocation of external tools. Dynamic deploy-time failures cannot be easily undone: scripts can alter the whole system (on purpose) and any non-trivial property about them is undecidable (the language is Turing complete and difficult to treat formally [36]), it is therefore impossible to determine before the execution of scripts which part of the system will be affected by their execution. This kind of failures has not been addressed by state of the art package managers.

Undetected failures are failures that remain undetected through upgrade deployment: according to all involved tools, the upgrade has been completed successfully, but the obtained system configuration contains incoherences. Undetected failures are the most subtle kind of upgrade failures, and can take very long (days, or even weeks) before being discovered, if they ever are. They can sometimes be fixed by configuration tuning (e.g. changing configuration keys or values to match the requirements of new sofware versions), but when this is not the case, an unsupported and error-prone manual rollback is the only solution left. As an example of such a failure, consider again Example 1 and imagine that `postinst` does not disable the `php5` module (not an unlikely scenario, given that such snippets are often written by hand). After removing `php5`, an incoherence is created in the system, as the `Apache` configuration still references a module which is no longer there. As `Apache` is *not* restarted upon removal of the module, the error will go

unnoticed during the upgrade. It will show up at the next `Apache` reload, which can take place months later, when the sysadm will most likely find out that `Apache` cannot be restarted, and will hardly relate it to the past upgrade. Addressing dynamic and undetected failures is hence a major issue for FOSS distributions.

Current package managers are able to detect only *static deploy-time failures*. The objective of EVOSS is to propose an approach able to detect also *dynamic deploy-time failures* and (current) *undetected failures*. To this aim EVOSS provides a *simulator* to predict the effect of maintainer script executions (see Section 3), to deal with deploy-time failures, and a *failure detector* component, which is able to deal with undetected failures.

3. Using models to enhance package upgrades

To improve the failure prediction of FOSS system upgrades, we propose a model-driven engineering (MDE) approach [2] called EVOSS (a preliminary version of the approach can be found on [7]) which relies on a model-based representation of the current system configuration and of all packages that are meant to be upgraded. This enables EVOSS to *simulate* upgrades as model transformations *before* upgrade deployment. To this end, we encode fine-grained configuration dependencies and abstract over maintainer scripts. This way the models capture all the information needed to anticipate the inconsistent configurations that current tools cannot detect, as they only rely on package metadata. The abstraction of maintainer scripts is realized by defining a new domain specific language as described in Section 4.

An overview of EVOSS is sketched in Figure 3. The simulation of a system upgrade is performed by the *Upgrade Simulator* which takes a set of models as input produced by the *Injector*: a *System Configuration Model* and *Package Models* corresponding to the packages which have to be installed/removed/replaced. The output of *Upgrade Simulator* is a new *System Configuration Model* if no errors occur during the simulation, otherwise an *Incoherences Report* is produced. The new *System Configuration Model* is queried and analyzed by the *Failure Detector* component. When *Failure Detector* discovers inconsistencies they are collected in the *Incoherences Report*. The real upgrade is performed on the system only if the new system configuration model is coherent.

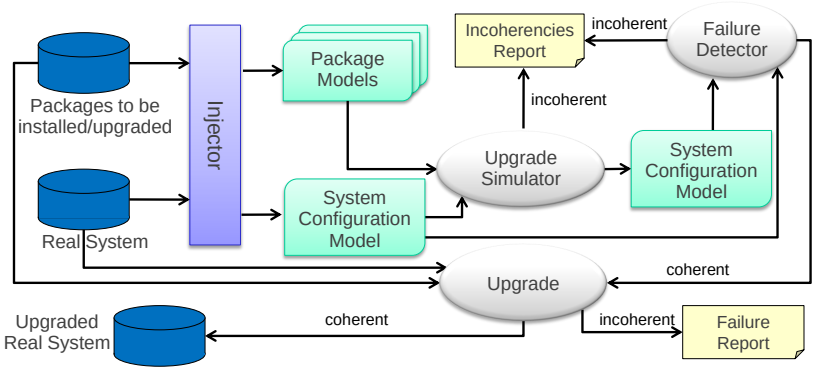


Figure 3: Overview of the EVOSS approach

The *System Configuration Model* describes the state of a given system in terms of installed packages, running services, configuration files, etc. The *Package Model* provides information about all packages involved in the upgrade, including maintainer script behaviour. The abstraction provided by EVOSS is more expressive than current package metadata. In fact, the proposed models also capture configuration dependencies and provide a representation of the maintainer scripts which have to be executed, as shown in Figure 4.

package	(c)	*file bundle*	$\supseteq$ *configuration files*	static
			$\supseteq$ *maintainer scripts*	dynamic
	(d)	***model***	$\supseteq$ *inter-package relationships*	static
			$\supseteq$ *configuration dependencies*	static
			$\supseteq$ *maintainer scripts representation*	dynamic

Figure 4: Modeling packages

The modeling constructs which can be used to define system and package models are defined in the *system configuration* and *package* metamodels, respectively. Such metamodels have been obtained by analyzing the FOSS system domain and by formalizing it, according to an iterative process consisting of two main steps: (a) elicitation of new concepts from the domain to the metamodel, and (b) validation of the formalization via instantiation to real systems. This process may be iterated further in case we discover limitations in the failure prediction power of the approach. These two metamodels are only briefly outlined in the rest of the section, but the complete metamodels can be found on the web at: `http://www.mancoosi.org/software/mancoosi-metamodels.tar.gz`.

System configuration metamodel. Figure 5 shows a fragment of the system configuration metamodel which contains the main concepts of FOSS system configurations. In particular, the `Environment` metaclass enables the specification of loaded modules, shared libraries, and running processes.

All system services can be used once the corresponding packages have been installed (note the association between the `Configuration` and `Package` metaclasses) and properly configured (`PackageSetting`). Moreover, the metamodel allows the configuration of an installed package to depend on other package configurations: this is a fundamental feature

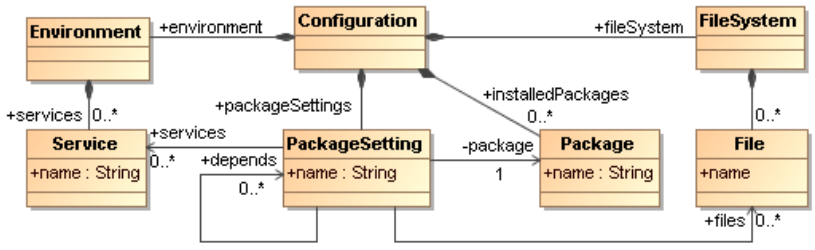


Figure 5: Fragment of the Configuration metamodel

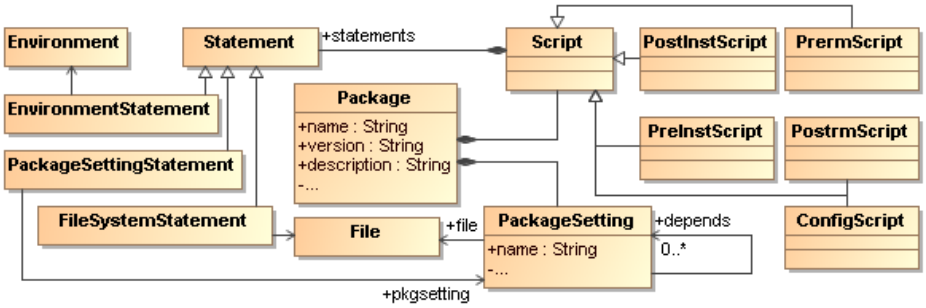


Figure 6: Fragment of the Package metamodel

that allows us to capture undetected failures, like the one that would arise if modifying the scripts of Example 1.

Package metamodel. The metamodel shown in Figure 6 contains the modeling constructs used to describe relevant package elements. In order to describe maintainer scripts behavior, the package metamodel contains the `Statement` metaclass which 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.

Inside the `Statement` metaclass, we provide an abstract representation of maintainer scripts which takes the form of a new domain specific language. We recall that "real" scripts are usually written in POSIX shell language and that all non-trivial properties about them are undecidable, including determining a priori their effects, for the purpose of reverting them upon failure. In this respect, a new language is required to reliably specify and simulate the behaviour of the scripts which are executed during package upgrades. In an ideal future scenario, all maintainer scripts will be written in the new DSL. In the meantime, in order to enable the application of the proposed upgrade simulation, specific support is required to translate the existing scripts in statements of the DSL as discussed in Section 3.1. Section **??** describes the upgrade simulator component, Section **??** describes the failure detector component, while Section 3.4 discusses the integration of EVOSS with legacy environments.

3.1. Model Injector

The first step to apply the proposed simulation approach is to build the system configuration and package models. In MDE terminology we need *model injectors*, apt to extract models from existing artifacts. EVOSS uses a specific model injection architecture that is implemented by using the Eclipse Modeling Framework (EMF)[1]. As shown in Figure 7, this architecture has a layered structure. The most specialized layer, which is in charge of querying the concrete system configuration, is distribution-specific and needs to be re-targeted to each new distribution; the other two layers are distribution-independent. The *Mancoosi Model Management* consists of Java code which provides

[1]Eclipse Modeling Framework: `http://www.eclipse.org/emf`

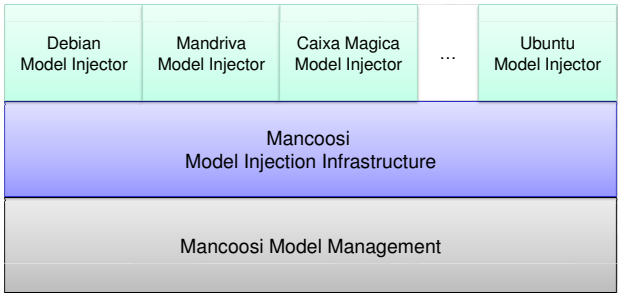


Figure 7: The Evoss model injection architecture

the infrastructure necessary to manipulate models conforming to the Configuration and Package metamodels shown in Figure 5 and in Figure 6. The *Mancoosi Injection Instractucture* contains a set of Java classes devoted to gather specific aspects of Linux distributions, like the file system, packages, alternatives, etc. This layer is distribution-independent and makes use of distribution-dependent injectors defined in the uppermost layer in Figure 7.

The outcome of the system injection is a model that represents, in a homogeneous form, different aspects of a running system, such as installed packages, users and groups, mime type handlers, alternatives, implicit dependencies, etc. Figure 8 shows a configuration model obtained as output of the model injector. This model consists of an environment composed of the services `sendmail`, and `www` (see the instances `s1` and `s2`) corresponding to the running mail and web servers, respectively.

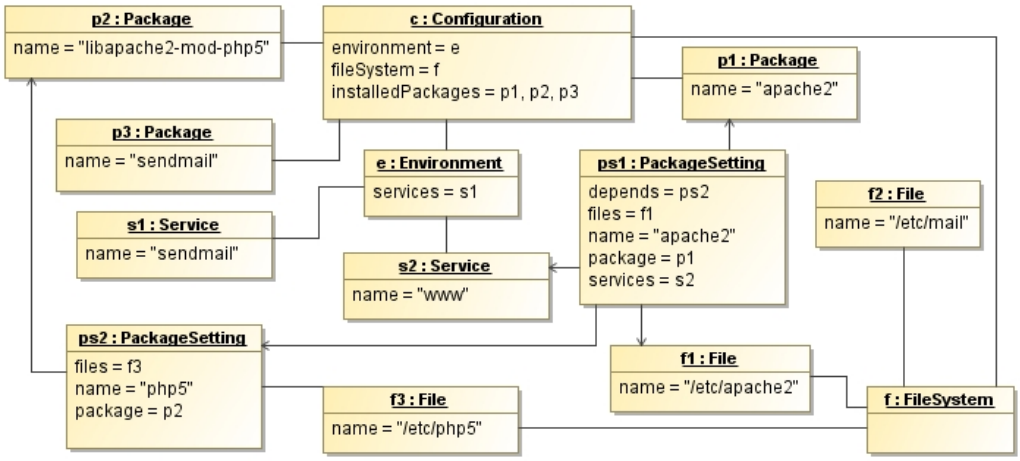


Figure 8: Configuration model: an example

The instances `ps1` and `ps2` of the metaclass `PackageSetting` represent the settings of the installed packages `apache2` and `libapachemod-php5`, respectively. The former depends on the latter (see the value of the attribute `depends` of `ps1` in Figure 8) and both are associated with the corresponding files which store their configurations. Such a

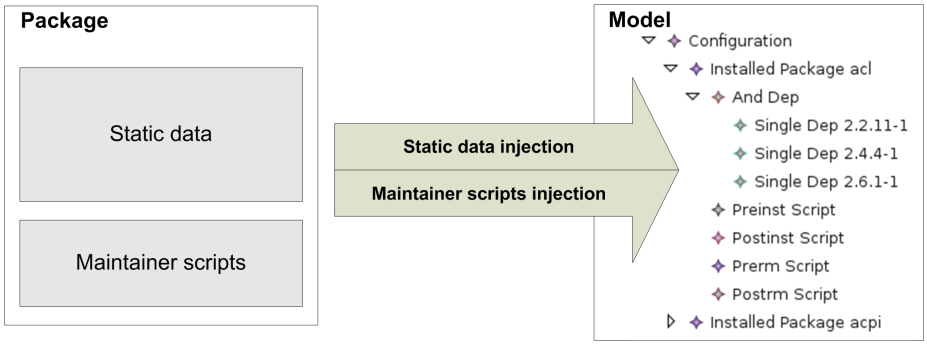


Figure 9: Overview of the package injection procedure

fine-grained, installation-specific dependency is not currently expressible using package metadata only.

A particular attention is required for the injection of packages since they have both static and dynamic parts. The outcome of package injection is shown in Figure 9, where the distinction between static and dynamic package parts can be appreciated. The resulting model contains modeling elements encoding both the considered package and its scripts (as DSL statements). The maintainer script injection requires specialized techniques and tools. We used Gra2MoL [5] which is a language especially tailored to specify mappings between grammar elements and target metamodel elements. A Gra2MoL transformation definition consists of rules transforming grammar elements into model elements.

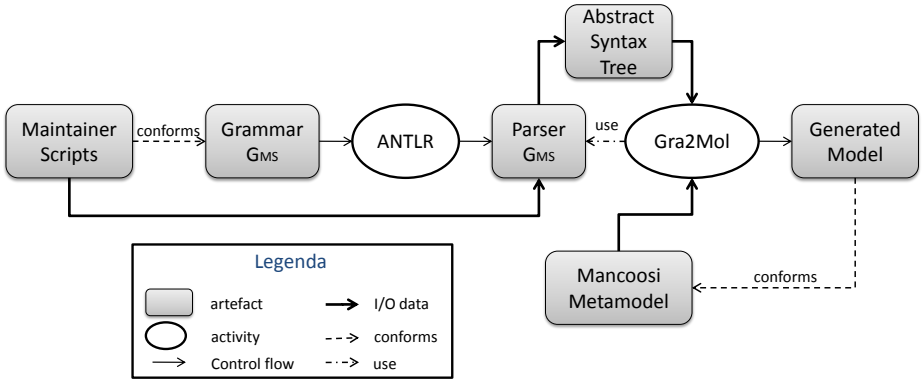


Figure 10: Maintainer scripts injection

The maintainer scripts injection process is depicted in Figure 10: G_{MS} is the grammar we defined for parsing the maintainer scripts. By means of ANTLR[2] we produced a parser for G_{MS}. The parser takes as input the maintainer scripts and produces an abstract syntax tree for the parsed scripts. The abstract syntax tree is taken as input by

[2] http://www.antlr.org

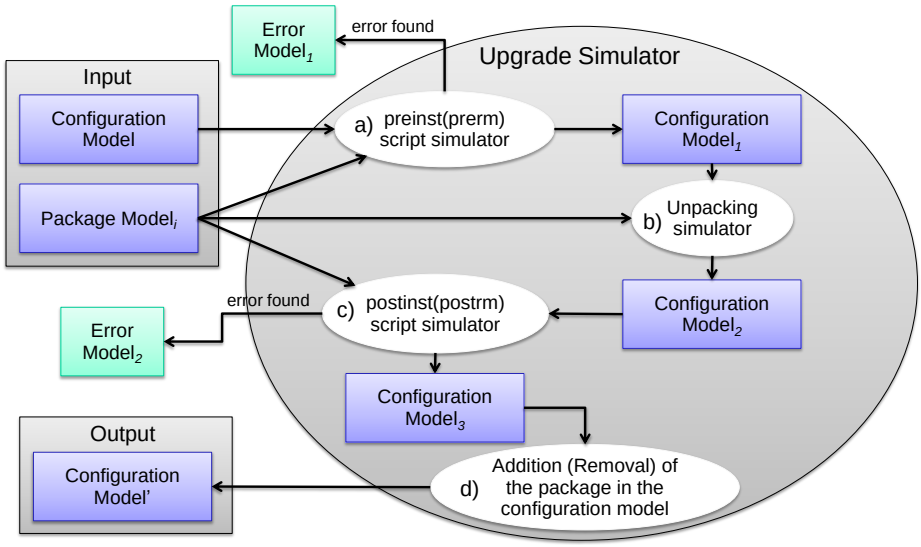


Figure 11: Upgrade simulator

Gra2MoL transformations which query it and generate target models. Further details about the whole injection process can be found at [16].

3.2. Upgrade Simulator

The simulation of system upgrades is performed by the *Upgrade Simulator* component shown in Figure 3. Given an upgrade plan consisting of a sequence of packages to be installed/removed/replaced, the corresponding package models are retrieved by means of the model injector. In particular for each package involved in the upgrade, the maintainer scripts injector retrieves the maintainer scripts associated to the package and transforms them in DSL scripts. Then, for each package model the simulator executes the four steps shown in Figure 11. Starting with the simulation of pre-install(-removal) scripts (step a), if no errors are identified, a new configuration model is obtained and the unpacking simulation is performed on it (step b). Then post-installation scripts are simulated on the obtained model (step c), and if no errors are encountered yet, the target configuration model is finalized by adding/removing/replacing the representation of the involved packages (step d). When an error is encountered during the simulation, specific error models are produced; they can be further queried and analyzed to better understand the nature of the error.

The most delicate part of the overall process is the simulation of the scripts which are executed during the upgrade. Figure 12 shows an overview of the simulator which has been conceived for such a purpose. The script simulator performs three subsequent activities. Given a maintainer script expressed in the DSL and composed of n statements $St_1, St_2, \ldots, St_n$, the activity a), namely *Retrieval of Model transformations*, retrieves, from the *Repository of Model transformations*, the model transformations associated to the n statements composing the script. It is important to recall that the model transformations provide the (operational) semantics of the script statements. More precisely,

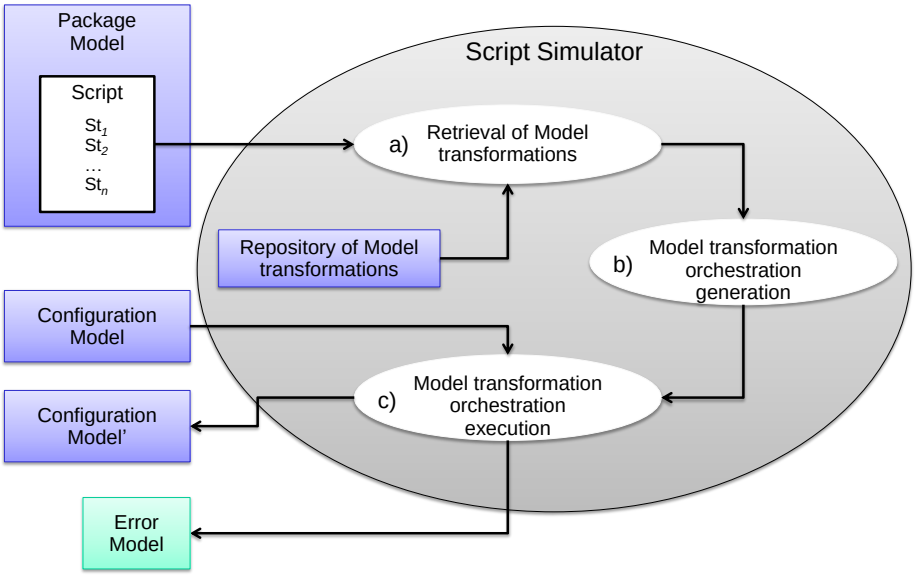


Figure 12: Script simulator

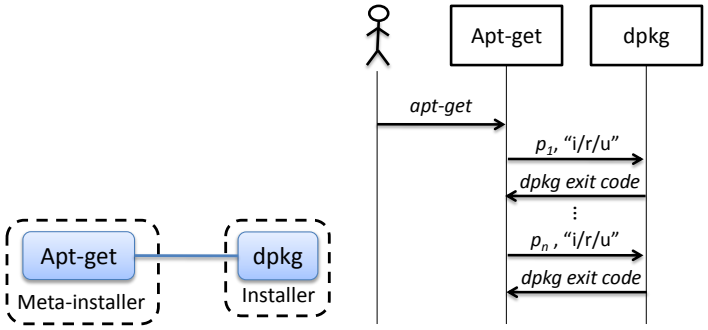
the model transformations define how a source configuration changes when DSL statements are executed. The activity b), namely *Model transformation orchestration generation*, properly chains these model transformations. Finally, the activity c), namely *Model transformation orchestration execution*, executes the chain of transformations on the source configuration model and, if no error is encountered, a new configuration is generated.

3.3. Failure detector

Given the current configuration model, the system upgrade is simulated by taking into account the packages that have to be upgraded. A *failure detector* is then used to check system configurations for incoherences. The coherence of a configuration model is evaluated by means of queries which are embodied in the failure detector. In particular, for each detectable failure, a corresponding OCL[3] expression is defined and used to query models and search for model elements denoting failures.

OCL is a declarative language that provides constraint and object query expressions on models and meta-models. A sample OCL query is shown in Listing 1: given a configuration model (`IN` conforming to the metamodel `MM`), all the instances of the `MimeTypeHandler` metaclass are retrieved and those which do not have a handler specified are considered. When the result of this query is greater than 0, an inconsistent configuration has been detected. These configurations are considered to be inconsistent due to the existence of mime types without corresponding handlers installed (e.g. the owning package has been deleted without un-registering the handler).

[3]OMG Object Constraint Language (OCL): http://www.omg.org/spec/OCL/



(a) Static view (b) Dynamic view

Figure 13: The *apt-get* meta-istaller: current solution

Listing 1: Fragment of the OCL query to detect missing mime type handlers

```
MM!MimeTypeHandler.allInstancesFrom('IN')
      ->select(e |
                  e.handler.oclIsUndefined()
               )->size()
```

The failure detector is extensible in the sense that when new incoherences are identified, a corresponding OCL query can be defined and added to it. If incoherences are identified in either the source or target configuration models, the upgrade cannot be performed; the sysadm can then either fix the problems manually or avoid deploying the upgrade.

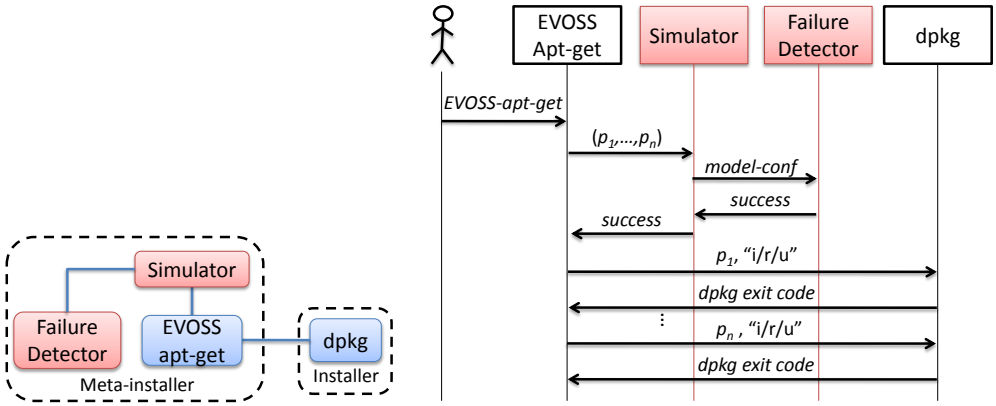
3.4. Integrating EVOSS *with legacy environments*

The integration of EVOSS with legacy environments is realized by suitably extending existing meta-installers. Figure 13 shows both the static and dynamic views of the popular meta-installer *apt-get*. Its architecture is very similar to other meta-installer architectures. As shown in Figure 13(b), *apt-get* receives the upgrade request from the user, plans the upgrade to be performed, retrieves the involved packages, and invokes *dpkg* for each package as needed. It is important to note that each kind of upgrade can cause either the installation, the removal or the upgrade of the involved packages. For each invocation of *dpkg*, *apt-get* checks the outcome of *dpkg* by means of its exit code.

Figure 14 shows the enhancement of existing meta-installers by means of the *Simulator* and the *Failure Detector* component provided by EVOSS. The upgrade request is sent to the *Simulator* component before performing the upgrade on the real system. Figure 14(b) shows the case in which both the *Simulator* and the *Failure Detector* components have a success as outcome. In this case the upgrade of the real system can be performed. In case the *Simulator* or the *Failure Detector* components detect errors, then the upgrade is stopped and the user is informed about the encountered problems.

4. Abstracting maintainer scripts

As discussed in Section 2.2, the main reason for unpredictability of failure upgrades is the difficulty to analyze maintainer scripts in their full generality. Our solution to



(a) Static view (b) Dynamic view

Figure 14: Meta-istallers enhanced by Evoss

improve predictability is to find a way to limit the expressive power of the *language* used in maintainer scripts, without reducing the *functionality* of the scripts themselves. Another possible way to improve the system reliability is to use testing approaches and in particular integration testing. However, while testing is able to spot failures only *after* upgrade deployment, Evoss aims at discovering failures *before* upgrade deployment.

To address this challenge, we need to first understand which part of the full-fledged scripting language is actually used in practice. We have therefore performed an extensive analysis of maintainer scripts used in two mainstream distributions: Debian[4], a very large distribution [20] based on the `dpkg` installer, and Fedora[5], based on the `RPM` installer.

This analysis allowed both the identification of recurring templates in maintainer scripts and the collection of them in a limited number of clusters. The adopted procedure and results can be found in [15] and we summarize here the relevant results: for Debian, we have analysed all the 25,440 maintainer scripts belonging to the Lenny release, discovering that 16,348 (64.3%) scripts are entirely generated by means of debhelper[6] templates, and only 9,061 (35.6%) contain snippets written by hand. After further investigations of the hand written snippets, we discovered a few additional templates, reaching a coverage of 66% maintainer scripts that can be *entirely* written using templates only. Concerning Fedora, we have considered all the available 2,038 maintainer scripts used in Fedora 11. The analysis has shown that 1,962 (93.6%) scripts are automatically generated starting from recurring templates. Fedora templates turned out not to be significantly different from Debian templates. At the end, we came up to 52 different templates [14]. Each one of those templates contains statements that are executed as a whole. Listing 2 shows a template example consisting of statements which get executed after the removal of

[4] http://www.debian.org

[5] http://fedoraproject.org

[6] Debhelper is a suite of utilities to streamline the maintenance of Debian-like packages. For more information please refer to http://packages.debian.org/sid/debhelper.

GNOME[7] components which ship GNOME configuration schemata.

Listing 2: Example of template

```
if [ "$1" = purge ]; then
    OLD_DIR=/etc/gconf/schemas
    SCHEMA_FILES="#SCHEMAS#"
    if [ -d $OLD_DIR ]; then
        for SCHEMA in $SCHEMA_FILES; do
            rm -f $OLD_DIR/$SCHEMA
        done
        rmdir -p --ignore-fail-on-non-empty $OLD_DIR
    fi
fi
```

Another example of templates is the management of *alternatives* which allow distribution and package owners to group and categorize packages that provide similar functionalities. For instance, different Java virtual machines installed on a same system are managed by means of an alternative named `java` which can point to a `java` executable among all the `java` executables registered within the alternative framework. This way the sysadm can freely chose which virtual machine will be launched by default. Due to the frequency of alternative management snippets, it would be profitably to have high-level statements to mimic alternative management *functionalities* during simulation.

We have then defined a DSL that captures all the recurring templates and contains limited control flow operations. The limited expressive power of the DSL is the price to be paid to have the DSL amenable to automated analysis. However, the DSL has also a tagging mechanism that allows us to specify the behaviour of script parts which cannot be completely specified with DSL statements. This way, script authors (usually package maintainers) can specify how such parts affect the configuration model and enable their simulation.

As usual for a programming language, our DSL has both abstract and concrete syntaxes. The abstract syntax of the DSL is part of the metamodel shown in Figure 6. The `Statement` metaclass represents an abstraction of the commands that can be executed

[7]GNOME: The Free Software Desktop Project - http://http:www.gnome.org

```
SCRIPT ::= STATEMENT_LIST

STATEMENT_LIST ::=
    PRIMITIVE_STATEMENT STATEMENT_LIST |
    COMPOUND_STATEMENT STATEMENT_LIST| ε

PRIMITIVE_STATEMENT ::=
    TEMPLATE_STATEMENT STATEMENT_LIST |
    OTHER_PRIM_STATEMENT STATEMENT_LIST | ε

COMPOUND_STATEMENT ::=
    CONTROL_STATEMENT STATEMENT_LIST |
    ITERATOR_STATEMENT STATEMENT_LIST | ε

TEMPLATE_STATEMENT ::=
    ALTERNATIVE | DESKTOP | DOCBASE | EMACS |
    GCONF | ICONS | INFO | INIT |
    LIBRARY | MENU | MIME | MODULES |
    SCROLLKEEPER | SGML | UDEV | USRLOCAL
    USERGROUPS | WM | XFONT

OTHER_PRIM_STATEMENT ::=
    ADDITION_STATEMENT STATEMENT LIST |
    DELETION_STATEMENT STATEMENT LIST | ε

ALTERNATIVE ::=
    rm_alternative(STRING,STRING) |
    add_alternative(STRING,STRING)
    ...

ADDITION_STATEMENT ::=
    addPackageSetting_dependences(STRING,STRING) |
    addEnvironment_runningServices(STRING,STRING) |
    ...| ε
...
```

Figure 15: Fragment of the DSL grammar

by a given script to affect the environment, e.g. the file system or the package settings of a given configuration. A fragment of the concrete syntax is shown in Figure 15: we can see that programs are built by composing *primitive* statements, which include both the 52 templates identified during script analysis and a few other primitive operations on the system model. All syntax details can be found in Chapter 4 of [16].

Besides simple command sequencing, compound statements can use a very limited set of control and iterator constructs. Control statements consist of a simple case instruction and a few instructions to abort script execution, but no other tests, jumps or loops are allowed. To counter the absence of general control flow operations, we provide various kinds of iterators, specialised on the data structures typically manipulated by maintainer scripts: (i) *directory iterator*, which enables to iterate on the files contained in a given directory, (ii) *line iterator*, which specifies iterations on the lines of a given file, (iii) *enumeration iterator*, which takes an enumeration iterator to work on an ordered set of indexes, (iv) *argument iterator*, which iterates through each command line argument, and (v) *word iterator*, which takes a string as input and repeats a set of statements for each word contained in the string.

Most of the code snippets found in maintainer scripts can be captured directly using the DSL constructs. Nevertheless there are still a few scripts containing commands that are not covered by templates represented in the DSL (as we remarked above, a tiny number of Fedora and a certain amount of Debian scripts are not covered by our 52 templates). To circumvent this limitation, we have included in our DSL a set of extra primitive statements (the `OTHER_PRIM_STATEMENT` in Figure 15), which allows us to describe arbitrary modifications of the system model, and can be used to capture the behavior of those commands that cannot be recoded in the DSL using the standard templates. These *other* primitive statements are classified into additions and deletions and can be used to specify the deltas [8] between two configuration models, encoding the semantics of script snippets that escape standard templates.

Example 2. *Let us suppose the maintainers write the following code:*

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

The current version of the DSL does not allow to directly recode this snippet, as we did not find it relevant enough to justify the need for a specific metaclass. Still, maintainers can specify its behavior as follows:

```
#<%
addPackageSetting_dependences(apache2,php5);
addEnvironment_runningServices(env,apache2);
#%if [ -e /etc/apache2/apache2.conf ] ; then
#%    a2enmod php5 >/dev/null || true
#%    reload_apache
#%fi
#%>
```

In Example 2, the maintainer specifies the behaviour of the script code by enclosing it in a `#<%...#%>` block. Immediately after `#<%`, a sequence of statements are given in order to specify how the configuration model changes if the considered script code is executed. Here, the execution of `a2enmod php5 >/dev/null || true` modifies the configuration model

by adding a new dependency between the package settings of the `apache2` and `php5` packages (line 2 above). Then `apache2` is reloaded and this entails the addition of `apache2` as a running service in the environment (line 3 above).

These specific statements allow us to define rules capable of checking if the removal of a package leads to an inconsistent state. For instance, in this example, a rule can be defined to check if the removal of the `php5` package does not forget to disable the `php5` in the Apache configuration.

This class of statements is intended to ensure full compatibility with existing distributions, and ease translation of maintainer scripts to the DSL. During the transition, the set of templates will grow to accommodate all the needs of package maintainers. As soon as all maintainer scripts will be written in terms of templates, it will be possible to remove these special statements from the DSL.

The semantics of the DSL is specified in an operational way in terms of model-to-model transformations (given in ATL[8]). For each command, a corresponding transformation is given to describe the exact command behavior when executed on the source configuration. In particular, each statement consists of a precondition to be evaluated on the source configuration and of a model transformation to be performed if the precondition is satisfied to produce an updated configuration (e.g., see Example 3).

Example 3. *The semantics of the statement* `rm_alternative` *is shown in Listing 3: the execution of the* `rm_alternative(name, location)` *removes the executable in* `location` *from the alternative* `name`*. If the alternative being removed does not exist in the source configuration, an error is raised (see lines 3-5), otherwise the alternative is not copied to the target configuration model.*

Listing 3: Fragment of the `rm_alternative` semantics

```
1  rule rm_alternative(name, location) {
2    do {
3      if (INConfiguration!Alternative.allInstances() -> select(a | a.name =
           ↪name) -> isEmpty()) {
4        'ERROR: The alternative' + name + ' does not exist'.println();
5      } else {
6        -- The action block is empty, no action is executed and hence
7        -- the Alternative name is not copied to target configuration
8      }
9    }
10 }
```

5. Experimental validation

We have validated the EvOSS approach by applying it on Fedora and Debian-based systems consisting of ≈1400 installed packages. After the automatic model generation from the running system (see Section 5.1), a static and dynamic analysis has been performed on the obtained models (see Section 5.2). The experiment shows the feasibility of the approach and how it is able to discover upgrade failures belonging to the failure classes discussed in Section 2.

[8] Atlas Transformation Language: http://www.eclipse.org/m2m/atl/

5.1. Injection of Debian-based systems

To implement an injector for supporting a new Linux distribution, one should start from the infrastructure presented in Section 3.1, extend the provided classes, and implement their abstract methods. According to such recipe, we implemented the Ubuntu 9.10 injector. This module is also able to retrieve information not directly available, which needs to be identified by means of a complex navigation on the real system. The identification of the dependencies between the configuration files is an example.

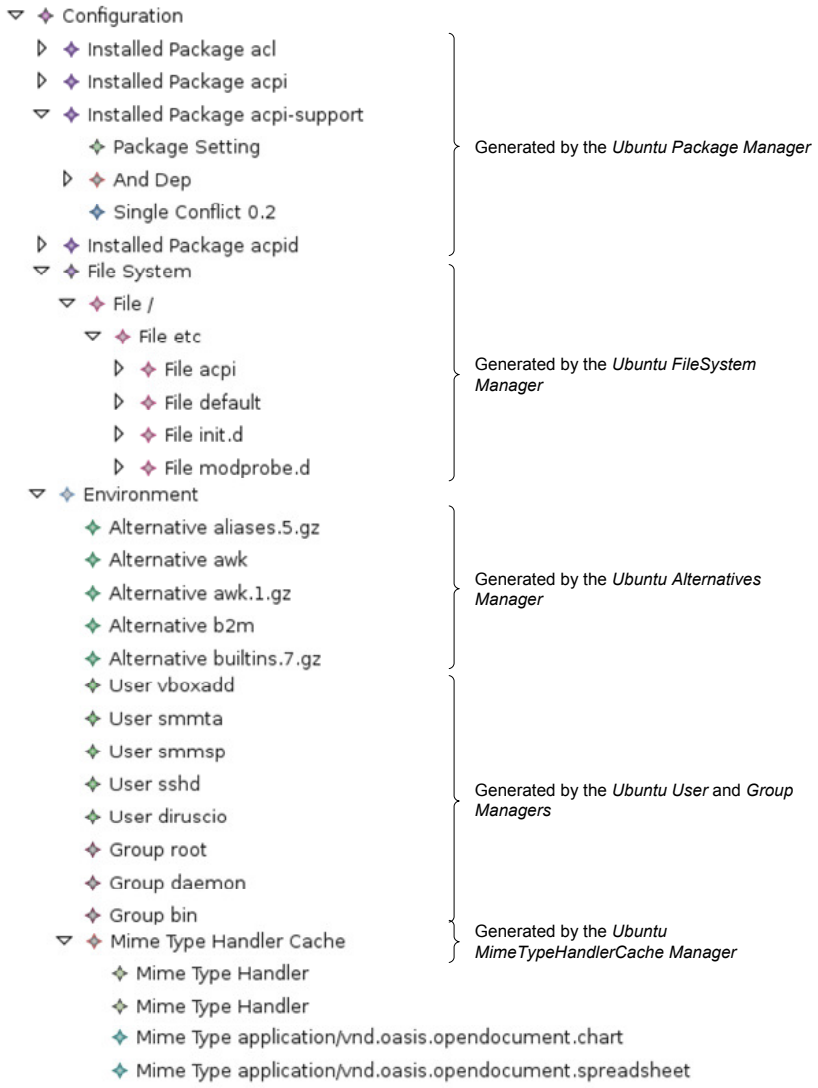


Figure 16: Fragment of an injected Ubuntu configuration

Figure 16 reports a small fragment of a configuration model generated by executing

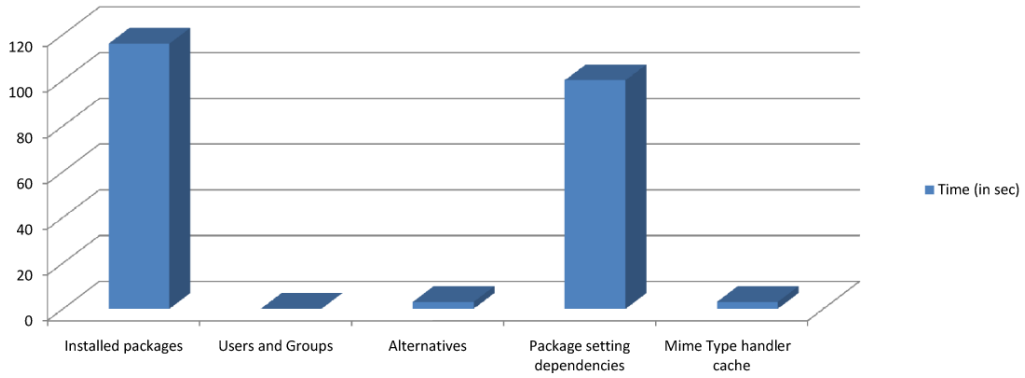


Figure 17: Timing injection of an Ubuntu 9.10 system on a test machine

the Ubuntu 9.10 injector on a running system. The model is shown by using a tree-based editor which is part of the Evoss model management layer.

The whole injection process has been tested on two different machines: a first experiment has been performed on an Ubuntu 9.10 system running on a machine with a quad-core processor, 2.83Ghz, 4GB of RAM with ≈1·400 installed packages. The whole injection process and the execution of the failure detector described in Section 5.2.2, have been completed in less than 240 seconds, as shown in Figure 17. Another experiment has been performed on an Ubuntu 9.10 system, running on a virtual machine with 512MB of RAM upon Virtual Box[9]. In this case the process took longer due to the limited hardware resources. It has been completed nevertheless in ≈1000 seconds.

5.2. Dealing with upgrade failures

In this section we show how Evoss is able to detect some typical upgrade failures belonging to the classes presented in Section 2.2, which can not be detected by means of currently available tools. In particular, Section 5.2.1 shows an example of dynamic deploy-time failure which can be detected on a Fedora 11 system by means of the Evoss *simulator*, while Section 5.2.2 sets out some examples of undetected failures which can be detected by means of the Evoss *failure detector* on an Ubuntu 9.10 system.

5.2.1. Dynamic deploy-time failures

Dynamic deploy-time failures can be detected by means of the Evoss *simulator*. We perform the simulation by executing each DSL statement on the configuration model, the outcome is an updated configuration model. Before executing the model transformation corresponding to the statement being executed, a transformation guard is evaluated; an error is raised if the check fails and the simulation is stopped.

Example 4. *When upgrading from the Debian* `Woody` *to* `Lenny` *a failure can occur due to the package* `tetex-bin`*. In particular, the* `Woody` *version of* `tetex-bin` *used to register the* `oxdvi.bin`

[9] http://www.virtualbox.org/.

alternative whereas the later version did not. To clean up the old alternative, upon upgrade, the maintainer uses the following `postinst` *snippet:*

```
case $action in
 configure|reconfigure)
   # upgrade from woody
   # since oxdvi is now integrated into xdvi, remove the alternative
   update-alternatives --remove-all oxdvi.bin
...
```

The alternative is removed without checking if it were registered; if it is not (e.g. when installing `tetex-bin` *from scratch) the script will fail.*

We are able to detect the failure in Example 4 thanks to the precondition of the `rm_alternative` DSL statement (see Example 3).

5.2.2. Failures undetected by existing approaches

In this section we show how the *failure detector* component of EVOSS is able to check the coherence of a running system by checking the corresponding configuration model with respect to some possible failures.

Implicit dependencies among packages. These are dependencies that are not declared in package meta-information but occur because of implicit dependencies between configuration files belonging to different packages. These become explicit in the configuration models and consequently can be easily discovered by means of simple OCL expressions. For instance, Listing 4 reports the OCL helper `isImplicitDependence` which returns `true` if the input package settings have a dependency between them, `false` otherwise. The detection of implicit package dependencies relies on this helper which is invoked for each pair of installed packages.

Example 5. *By considering the configuration model in Figure 8, the dependence between the* `apache2` *and* `php5` *package settings is detected even if the corresponding packages do not depend explicitly on each other as described in Section 2. In fact there is no dependence between the package elements* `p1` *and* `p2` *shown in Figure 8.*

Listing 4: Fragment of the OCL query to detect implicit package dependencies

```
helper def : isImplicitDependence(ps1:PackageSetting, ps2:PackageSetting):
    ↪Boolean =
  if ps1.depends->includes(ps2) or ps2.depends->includes(ps1) then
    true
  else
    false;
```

In EVOSS the detection of such dependences is very simple since it consists of querying the configuration model and checking the existence of references between package setting elements. In current distributions, the available tools are unable to extract such dependencies between configuration files. In the following we describe some of the failures that can be currently detected by EVOSS:

Missing configuration files. The system configuration model can easily identify the configuration files that are required, but not actually available in the system. For instance in Figure 18 the file `/etc/ldap/ldap.conf` is required by the installed package `libldap-2.4-2` but it is actually missing in the injected configuration model;

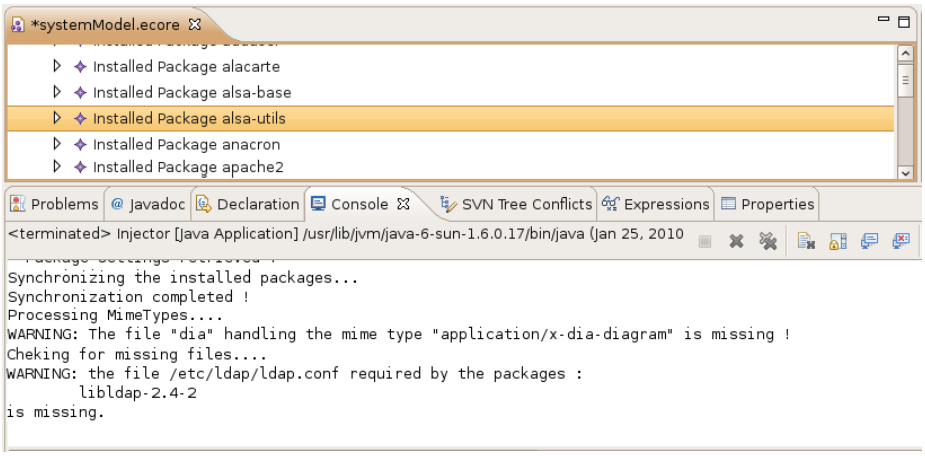


Figure 18: Some failures detected by static analysis of a system configuration model

Mime-type dangling handlers. The models allow us to identify the mime types that the system should be able to manage, but whose handler is missing from the system. For instance, in the configuration considered in Figure 18 the executable "dia" handling the mime type "application/x-dia-diagram" is missing, while according to the configuration model this mime type should be manageable;

Missing services. The `/etc/init.d` directory contains all services that are available on a Linux system. The `/etc/rc*.d` directories specify which services should be started in which "runlevel". These queries identify services that are not configured to be active in the current system.

6. Discussion

In this section we provide some considerations on Evoss, revolving around three topics: community support, flexibility, supporting tools, and performance.

Community support. We are aware that the failure detection abilities of Evoss are not complete with respect to all possible upgrade failures; what has been presented here is rather just a representative subset of failures that have been found to be common during package upgrades. We hope for a community participation in the iterative and probably never ending improvement of failure detection abilities. In fact, adding a new class of detectable errors is relatively straightforward in Evoss: it is enough to add the corresponding OCL queries to the failure detector, and queries turn out to be quite concise. Even though OCL is not a language with which sysadms are necessarily familiar, similar experiences (such as QWG templates in Quattor [6]) show that, once the usefulness of a tool has been proven, the interest of the sysadm community easily grows around it, even when the learning curve is steep.

Evoss *adoption and flexibility.* We have shown how Evoss represents a clear advancement in dealing with reliable upgrades for FOSS distribution installations. The price for this advancement includes the effort required to integrate Evoss in real environments and the loss of flexibility when compared to a maintainer being able to use a general scripting language for maintainer scripts.

By focusing on the integration of Evoss in real environments, two aspects should be considered: (i) in Section 3.1 we introduced the model injector that promotes the use of Evoss in practice since thanks to this module the needed models (included DSL statements representing the maintainer scripts) are automatically extracted from the actual system; (ii) as discussed in the previous item we hope that the community of users will be involved in the definition of common errors thus increasing the capabilities of the failure detector.

Regarding the loss in flexibility it is interesting to note that from the analysis we performed we realized that often maintainers do not really need a full-fledged language as POSIX shell to write maintainer scripts. Nevertheless, in some cases they do need more flexibility. For this reason we added the tagging mechanism (see Section 4) that allows maintainers to write more complex scripts by using any language. While minimal knowledge about models and their manipulation is needed to do that, maintainers are only asked to add tagging information about changes that scripts make on a configuration when executed.

To sum up, Evoss is flexible since (i) it allows maintainers to write scripts with traditional scripting languages or by means of DSL statements; (ii) the tagging mechanism of the DSL is another dimension of this flexibility allowing the definition of complex scripts.

Status of the Evoss *supporting tools.* The Evoss supporting tool consists of different components: (i) the system configuration injector, (ii) the package (including the maintainer scripts) injector, (iii) the DSL and the simulator, and finally (iv) the failure detector. In the following we explain the implementation status of each component:

1. *System configuration injector*: this part is complete with respect to the system configuration concepts that have been considered for defining the DSL and its simulator. This means that a refinement of the DSL (performed for example to better cover the set of existing scripts) will probably require a revision of the metamodels and then of the system configuration injector;
2. *Package (including the maintainer scripts) injector*: we have a prototype of this component that makes use of the Gra2MoL language [5]. It is in an alpha state: for more details please refer to the deliverable D2.2 [13];
3. *Mancoosi DSL and Simulator*: we have a prototype of the DSL and of the simulator components. Even though these components are in a more advanced state than the package injector, we are still testing them on real distributions;
4. *Failure detector*: the failure detector is complete with respect to the failures mentioned in Section 5.2.2.

The supporting tools provided with a working Eclipse bundle can be found at `http://www.mancoosi.org/software/evoss.tar.gz`. It consists of the components 1 and 4. However, according to the description of work of the Mancoosi project all the components

previously described have to be finalized and integrated with existing Linux distributions such as Debian[10] and Caixa Mágica[11] by the end of the project.

Performance of Evoss. Our empirical tests show that simulation and failure detection do not add significant overhead to the upgrade process (which is anyhow usually dominated by other factors, such as download times). The most expensive task of Evoss is model injection. The performance of this module when injecting an Ubuntu system is reported in Section 5; injection takes less than 4 minutes. However, it is important to note that the measurement represents the performance required to build models from scratch. Typically the model update is performed by means of simulation which modifies models accordingly to the real system. However, it is always possible for the model and the real system to be slightly out-of-sync. For instance, we cannot forbid a user to manually delete files or to manually change configuration aspects. In those cases the model injection must be performed again. We are currently investigating optimizations of the injection performance by exploiting the existent configuration model and doing "diff"-updates.

7. Related work

Configuration management research has already faced the challenge of maintaining and deploying the configuration of (*nix) machines over time. Several approaches and Configuration Management Tools (CMTs) exist to that end, the most relevant ones being: Bcfg2 [11], LCFG [1], PoDIM [10], Quattor [6], and Puppet [23, 22]. A widespread recent trend among CMTs is the use of declarative configuration languages (as opposed to what pioneers like cfengine [4] used to offer); such languages offer in some cases the ability to express finer grained dependencies than those of FOSS packages. Nonetheless, CMTs seem to invariably suffer from two important shortcomings with respect to our approach:

1. The sysadm has to specify explicitly, by means of the declarative language, all the aspects of a system configuration. The possibility to spot implicit dependencies is nullified, since only the specified configuration aspects will be checked. Additionally, model injection facilities provided by Evoss enable the detection of those dependencies which are not appreciated by the sysadm. That makes Evoss more suitable for regular user machines, where CMTs are rarely used and users often lack system administration skills.
2. The second shortcoming is that existing CMTs are only able to grasp some of the static aspects of a system that might induce upgrade failures; such tools are not able to manage relevant dynamic aspects of maintainer scripts and hence cannot shield users from upgrade failures due to them.

An additional shortcoming is that the attention of such tools for configuration validation seems to be lower than what we propose with Evoss, where the key is validation. As a consequence, Evoss appears to be more easily extensible with new failure predicates (it only requires adding new OCL queries) than existing CMTs.

[10] http://www.debian.org
[11] http://www.caixamagica.pt

In spite of the above differences, there are potential synergies between Evoss and declarative CMTs. In particular, when the configuration described in CMTs can be trusted[12], model injection can be extended by grasping new dependencies that are known to the CMT.

Letting aside CMTs, we are aware of four contributions which are more specifically comparable to Evoss. Conary [21] is a system that manages package upgrades via automated dependency resolution against distributed on-line repositories. To replace maintainer scripts Conary introduces the notion of dynamic tags, which are analysed to detect similar operations performed by package installation, and groups them together. In the end, these operations are performed on the system, just like maintainer scripts, and the issue of predicting and preventing failures is still there.

McQueen [28] proposed to instrument the upgrade process to monitor the files that are actually being modified, to be able to restore them in case of failures. Unfortunately, it is not always possible to undo an upgrade by simply restoring the old copy of all touched files, and it is necessary to consider also the system configuration, the running services, etc., as taken into account by our metamodels. This kind of tools can be ideally combined with Evoss to detect discrepancies between the model and the actual system. That can be done by comparing execution traces of the DSL with the actual maintainer scripts. Each discrepancy will lead to refining the DSL, hence improving the simulation provided by Evoss.

NSIS[13] is an open source system to build auto-installers for Windows. NSIS recognizes the importance of providing specific primitives to write maintainer scripts, and makes some steps towards defining a useful domain specific language. However the language contains full-fledged conditionals, functions, labels, gotos, and arbitrary integer expressions, which make it Turing-complete, missing completely the advantages of our DSL.

Finally, NixOS [17] is a purely functional distribution, where static parts of a system (packages, configuration files, boot scripts, etc.) are built from pure functions. The approach promises to render upgrade failures irrelevant, as they can be easily undone. Unfortunately this is not always the case, as some operations can not be made purely functional (e.g. user database management).

Our work can also be related with techniques for static analysis of scripting languages, like [36] which deals with SQL injection detection for a limited subset of PHP scripts, or [27] which provides a mechanism to detect argument arity bugs in a limited subset of shell scripts. Both works clearly show the great difficulties one has in handling scripting languages in their generality, and cannot be reused in our framework.

Our approach can also be considered as a specific case of software modernization, where significant work has been done by the OMG via the Architecture Driven Modernization (ADM) task force [32] that aims at building standard metamodels and tools for supporting software renewal, and Reus et al. in [34, 19] propose similar MDA processes for software migration. Nevertheless, these works are mostly focused on building UML models, which is a very different focus from our goal.

[12] Note that here *trust* is a matter of correctness guarantees, not of security concerns.
[13] http://nsis.sourceforge.net/Main_Page

8. Conclusions and future work

We have discussed how current technology used to manage FOSS distributions leaves many types of upgrade failures undetected. We have then shown how to significantly enlarge the class of detectable failures by adopting a novel model-based approach. The approach—called Evoss—takes into account both fine-grained static aspects of system configurations (e.g. dependencies among configuration settings) and dynamic aspects of upgrades (e.g. configuration scripts and their effect on the system state). Both aspects are currently ignored by state of the art package managers. An essential component of Evoss is a DSL capturing all essential operations performed by configuration scripts, and yet is simple enough to be amenable to analysis and simulation.

This approach has been practically validated instantiating it on two widely used FOSS distributions. Evoss has been designed with continuous refinement in mind, with the explicit intention of involving the system administrator FOSS community in iteratively adding primitives to the DSL.

As a future research direction, we will investigate how Evoss and suitable transactional logs can be used to drive at run-time the roll-back of residual effects of failed or undesired upgrades.

References

[1] Anderson, P., Scobie, A., 2000. Large scale Linux configuration with LCFG. In: ALS'00: Proceedings of the 4th annual Linux Showcase & Conference. USENIX, pp. 42–42.

[2] Bézivin, J., 2005. On the unification power of models. SOSYM 4 (2), 171–188.

[3] Brown, A. W., Wallnau, K. C., 1998. The current state of CBSE. IEEE Software 15 (5), 37–46.

[4] Burgess, M., 1995. A site configuration engine. Computing Systems 8 (2), 309–337.

[5] Cánovas Izquierdo, J. L., Molina, J. G., 2009. A domain specific language for extracting models in software modernization. In: ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. Springer-Verlag, Berlin, Heidelberg, pp. 82–97.

[6] Childs, S., Poleggi, M. E., Loomis, C., Mejías, L. F. M., Jouvin, M., Starink, R., Weirdt, S. D., Meliá, G. C., 2008. Devolved management of distributed infrastructures with Quattor. In: LISA. USENIX Association, pp. 175–189.

[7] Cicchetti, A., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S., 2010. A Model Driven Approach to Upgrade Package Based Software Systems. Vol. 69 of chapter Communications in Computer and Information Science (CCIS). Springer, Heidelberg, pp. 262–276.

[8] Cicchetti, A., Di Ruscio, D., Pierantonio, A., 2007. A metamodel independent approach to difference representation. Journal of Object Technology 6 (9), 165–185.

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

[10] Delaet, T., Joosen, W., 2007. PoDIM: A language for high-level configuration management. In: LISA. USENIX, pp. 261–273.

[11] Desai, N., Bradshaw, R., Lueninghoener, C., 2006. Directing change using Bcfg2. In: LISA. USENIX, pp. 215–220.

[12] Di Cosmo, R., Trezentos, P., Zacchiroli, S., 2008. Package upgrades in FOSS distributions: Details and challenges. In: HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades. ACM, New York, NY, USA, pp. 1–5.

[13] Di Ruscio, D., Pelliccione, P., Pierantonio, A., Jan. 2010. Instantiation of the metamodel on a widely used gnu/linux distribution. Mancoosi Project deliverable D2.2, http://www.mancoosi.org/reports/d2.2.pdf.

[14] Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S., Jan. 2009. Metamodel for describing system structure and state. Mancoosi Project deliverable D2.1, http://www.mancoosi.org/reports/d2.1.pdf.

[15] Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S., 2009. Towards maintainer script modernization in FOSS distributions. In: IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems. ACM, New York, NY, USA, pp. 11–20.

[16] Di Ruscio, D., Thomson, J., Pelliccione, P., Pierantonio, A., Nov. 2009. First version of the DSL based on the model developed in WP2. Mancoosi Project deliverable D3.2, http://www.mancoosi.org/reports/d3.2.pdf.

[17] Dolstra, E., Löh, A., 2008. NixOS: a purely functional Linux distribution. In: ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming. ACM, New York, NY, USA, pp. 367–378.

[18] D'Souza, D., Wills, A., 1999. Objects, components, and frameworks with UML: The catalysis approach. Addison-Wesley, Boston, MA.

[19] Fleurey, F., Breton, E., Baudry, B., Nicolas, A., Jézéquel, J.-M., 2007. Model-driven engineering for software migration in a large industrial context. In: MoDELS'07.

[20] Gonzalez-Barahona, J., Robles, G., Michlmayr, M., Amor, J., German, D., 2009. Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14 (3), 262–285.

[21] Johnson, M. K., 2005. Building linux software with conary. In: Proceedings of the Linux Symposium.

[22] Kanies, L., 2003. ISconf: Theory, practice, and beyond. In: USENIX-LISA'03. USENIX Association, pp. 115–124.

[23] Kanies, L., 2006. Puppet: Next-generation configuration management. ;login: the USENIX magazine 31 (1), 19–25.

[24] Lehman, M. M., Belady, L. A. (Eds.), 1985. Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA.

[25] Lehman, M. M., Ramil, J. F., 2000. Software evolution in the age of component-based software engineering. IEE Proceedings - Software 147 (6), 249–255.

[26] Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R., Sep. 2006. Managing the complexity of large free and open source package-based software distributions. In: ASE 2006. pp. 199–208.

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

[28] McQueen, R., May 2005. Creating, reverting & manipulating filesystem changesets on Linux. Part II Dissertation, University of Cambridge.

[29] Mens, T., Demeyer, S., (eds.), 2008. Software evolution. Springer.

[30] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M., 2005. Challenges in software evolution. In: IWPSE '05.

[31] Naur, P., Randell, B., October 1968. Software engineering report on a conference sponsored by the NATO science committee. p. 231.

[32] (OMG), O. M. G., 2010. Architecture-driven modernization (ADM). http://adm.omg.org/.

[33] Raymond, E. S., 2001. The cathedral and the bazaar. O'Reilly.

[34] Reus, T., Geers, H., Deursen, A. V., 2006. Harvesting software systems for MDA-based reengineering. In: ECMDA-FA 2006. Vol. 4066. pp. 213–225.

[35] Treinen, R., Zacchiroli, S., 2009. Expressing advanced user preferences in componet installation. In: IWOCE'09. ACM.

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