mancoosi
managing software complexity

Model-based framework for managing the complexity and the state of the GNU/Linux instantiation

Deliverable 2.3

Nature : Deliverable
Due date : 23.05.2011
Start date of project : 01.01.2008
Duration : 40 months

SEVENTH FRAMEWORK PROGRAMME

Information Society Technologies

Specific Targeted Research Project
Contract no.214898
Seventh Framework Programme: FP7-ICT-2007-1

A list of the authors and reviewers

Project acronym	MANCOOSI
Project full title	Managing the Complexity of the Open Source Infrastructure
Project number	214898
Authors list	Davide Di Ruscio <diruscio@di.univaq.it> Patrizio Pelliccione <pellicci@di.univaq.it> Alfonso Pierantonio <alfonso@di.univaq.it> Roberto Di Cosmo <roberto@dicosmo.org> Stefano Zacchiroli <zack@pps.jussieu.fr>
Internal review	Anne-Sophie Refloc'h
Workpackage number	WP2
Deliverable number	3
Document type	Deliverable
Version	1
Due date	23/05/2011
Actual submission date	23/05/2011
Distribution	Public
Project coordinator	Roberto Di Cosmo <roberto@dicosmo.org>

Abstract

Modern software systems are moving towards an on-line mode of operation where long down-times due to maintenance problems are no more acceptable. With the shift towards an on-line mode of operation the "minimal acceptable standard" for the quality of service of modern software systems has been raised to a very high level.

One of the most challenging modern software systems are Free and Open Source Software (FOSS) distributions. Even big companies such as Google and Linden Lab, or large public bodies, like the French Ministry of Finance, base their information technology infrastructures on FOSS components. FOSS systems are typically based on fine-grained units of software deployment, called packages, which evolve in a non-centralized and controlled way and are frequently released [Ray01].

The complexity of deploying a complete FOSS infrastructure has led to the development of what are now called *distributions* (e.g., Mandriva, Ubuntu, Fedora, etc.), which are consistent and functional sets of software components released together with the software that is necessary to set up a complete operating system. This software includes also automated mechanisms for managing the packages the distributions are made of. These automated mechanisms, called package managers, make use of packages metadata in order to upgrade the distribution, i.e., to handle package installations, removals and upgrades. The most important information that is contained in packages metadata concerns the specification of dependencies (i.e., what a package needs in order to be correctly installed and to function correctly), and conflicts (i.e., what should not be present on the system in order to avoid malfunctioning). These packages are equipped with maintainer scripts which are executed before and after upgrades to perform configuration actions. Unlikely, by using existent package managers it is possible to easily make the system unstable by installing, removing, or upgrading some packages that "break" the consistency and coherence of what is installed in the system itself.

In the context of the Mancoosi project, a model-based framework called Evoss (EVolution of free and Open Source Software) [DCDRP+10] has been conceived to support the upgrade of FOSS systems. The approach promotes the simulation [DPD11] 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 package managers. 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 [DP11]. 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.

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 fault 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.

In this document, the *upgrade simulator* and the *fault detector* components of Evoss are described in details. Their application on the Debian distribution is also discussed.

Conformance

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

[1] http://www.ietf.org/rfc/rfc2119.txt

Contents

List of Figures

List of Acronyms

ACID Atomicity Consistency Isolation Durability

ATL ATLAS Transformation Language

CUDF Common Upgradeability Description Format

DSL Domain Specific Language

DUDF Distribution Upgradeability Description Format

FOSS Free and Open Source Software

GPL General Purpose Language

MANCOOSI Managing Software Complexity

MDE Model Driven Engineering

MOF Meta Object Facility

OMF Open Source Metadata Framework

OMG Object Management Group

POSIX Portable Operating System Interface [for Unix]

UML Unified Modeling Language

XML XML Metadata Interchange

Chapter 1

Introduction

Component-based software engineering is seen as a way for simplifying software development. A component-based software system is an assembly of software components, usually implemented by means of third-party libraries, designed to meet the system requirements [IC02]. Software development based on the use of existing components has origin in the 1960s [NR68] and practical applications have been widely explored in practice [BW98, DW99].

Modern software systems increasingly require to undergo continuing evolution [LB85]. The evolution management of a component-based system cannot neglect the numerous relationships [VR02], implicit or explicit, among components which might be affected when performing routine component management operations (e.g., additions, removals, or upgrades of components) as this can easily lead to unusable and corrupted systems.

A consequence of this continuous evolution is that software systems are expected to become more resilient, i.e., these systems are required to persistently deliver their services in a trustworthy way despite of changes. Despite the relevant body of research on software evolution (see e.g., [MDe08]), there is still the need for better predictive models which are able to anticipate the consequences of applying specific system changes [MWD+05]. Specifically, the evolution management of component-based systems is intrinsically difficult and requires techniques, algorithms, and methods which are both expressive and computationally convenient in order to be used in practice.

When maintaining large component-based software systems, it is essential to manage the upgrade in such a way that the good properties of the current system state are not disrupted. We intend upgrade as the possibility to modify an existing system by replacing some of the existing components with others, by adding new components or by deleting existing ones. There is extensive literature on component-based software evolution in many specific cases (e.g., [LR00, Vig01, BW98, DW99]); in the following we rather give a general abstract framework to describe some of the fundamental issues that component ugprade poses, independent of the specific component model or programming language involved. Then, we show how model-driven techniques can help on practically realizing the proposed abstract framework by using models, model transformations, and model elicitation techniques.

1.1 Component-based software evolution

At a very abstract level, a software system s can be seen as belonging to a set of states $\mathcal{S}$, equipped with a coherence relation $Coher : \mathcal{S} \rightarrow Bool$ that identifies those states that are *coherent*, according to some application-specific notion. A software unit u usually consists of a pair $\langle c, d \rangle$, with c the software component and d a description of it. The latter typically contains a brief description ranging from basic information, such as the unit name, to more structured specifications of the component behavior and its dependencies on other components, such as a model written in terms of some modeling language.

An upgrade operation $upgrade(u, op)$ involving a single unit u modifies the system by performing a particular operation op on u, such as addition, removal, or replacement by another version. Upgrades, and in particular the ones we are interested in, may involve programmatic steps, that can produce errors, so an upgrade is a system transformation of type:

$$upgrade(u, op) : \mathcal{S} \rightarrow \mathcal{S} \uplus Err$$

1.1.1 Successful upgrades

According to the specific target application, the notion of "successful upgrade" may vary: at the very minimum, one requires that an $upgrade(u, op)$ performed on an initial state s_i completes *without execution errors*, making the system evolve to a final state s_f. This is shown in the following Property 1.1.

$$\frac{upgrade(u, op)(s_i) = s_f}{s_i \mapsto_{u,op} s_f} \tag{1.1}$$

In general, one would expect more than that. A successful upgrade, as shown in Property 1.2, should only be one that makes the system evolve without errors to another *coherent* state.

$$\frac{upgrade(u, op)(s_i) = s_f \wedge Coher(s_f)}{s_i \mapsto_{u,op} s_f} \tag{1.2}$$

Additional conditions might be required on the state on which the upgrade is performed. Referring to Property 1.2 one might think to perform the upgrade only when the system is in a *coherent* state: $Coher(s_i)$. However, this is too restrictive since this abstract framework can be also used to perform the upgrade when the system is in an incoherent state; in this case the aim is to recover the system by performing some error recovery operation.

Ideally, one would also like to be able to check whether the upgrade will be successful *before* actually executing the upgrade, to avoid disrupting the system and all the hassle of rolling it back to a previous coherent state. Let us assume that there exists a precondition predicate pre that satisfies the Property 1.3.

$$\frac{pre((u, op), s_i)}{upgrade(u, op)(s_i) = s_f \wedge Coher(s_f)} \tag{1.3}$$

Then, it is then possible to ensure that the upgrade operation will be successful by just checking that the preconditions of Property 1.4 are satisfied.

$$\frac{pre((u, op), s_i)}{s_i \mapsto_{u,op} upgrade(u, op)(s_i)} \tag{1.4}$$

We notice that it is possible for a sequence of upgrades starting from a coherent state s_i and ending into an coherent state s_f, to pass through intermediate incoherent states. For example, if a functionality offered by one component u_0 is to be replaced by the same functionality offered by the combination of two components u_1 and u_2, removing u_0 and then adding u_1 and u_2 leads to a new coherent state, but all the intermediate states are broken. It is then necessary to perform this series of upgrade operations *as a whole*, and we write such operations:

$$upgrade(\overline{u, ops}) = \{upgrade_1(u_1, op_1), \ldots, upgrade_n(u_n, op_n)\}$$

1.1.2 Successful upgrades are not easy

In most component based deployment and maintenance systems it is possible to find some sort of precondition predicate implemented. However, in practice the precondition predicate *pre* previously introduced, is not computed on the full system state s and on the full upgrade operation $upgrade(u, op)$; it is only computed on abstract approximations of this information. Typically, this information consists of the description d of the modified component u, and of some limited part of the system state (e.g., the list of installed components, the file system, etc.). Thus, Property 1.4 is not easily achievable, as it implies:

1. the *correctness* of the description d with respect to u, as this information is being used to determine whether the preconditions is satisfied or not; and
2. the *soundness* of the precondition predicate *pre* with respect to the execution of the $upgrade(u, op)$ operation, which may contain arbitrary code.

1.2 Controlled upgrades via model transformation

To make progress towards establishing Property 1.4, we propose an iterative approach based on models (representing system abstractions) and simulations, that will allow us to build approximations of the precondition predicate *pre*.

A state $s \in \mathcal{S}$ can be *represented* by a model $Mod(s) \in \mathcal{M}$, where $\mathcal{M}$ is the set of all the possible system abstractions. Then, to each $upgrade(u, op)$ we can associate, using a mapping $\mathcal{A} : \mathcal{U} \to \mathcal{P} \times \mathcal{T}$, a *guarded* model transformation consisting of a pair (P, T) with $P \in \mathcal{P}$ a predicate and $T : \mathcal{M} \to \mathcal{M} \uplus Err$ a transformation capable of consistently transforming $Mod(s_i)$ into $Mod(s_f)$ or eventually fail. The following diagram shows the relations among the upgrades on states and their associated model transformations:

$$\begin{array}{ccc} Mod(s_i) & \xrightarrow{\mathcal{A}(upgrade(s,op))} & Mod(s_f) \\ Mod \uparrow & & \uparrow Mod \\ s_i & \xrightarrow{upgrade(u,op)} & s_f \end{array} \tag{1.5}$$

A *strong simulation* is then a mapping:

$$\sigma : \mathcal{M} \times (\mathcal{P} \times \mathcal{T}) \to \mathcal{M} \uplus Err$$

such that $\sigma(Mod(s_i), \mathcal{A}(upgrade(u, op))) = M_f$ implies $upgrade(u, op)(s_i) = s_f$, with $Coher(s_f)$ and $M_f = Mod(s_f)$ for any $upgrade(u, op) \in \mathcal{U}$ and $s_i \in \mathcal{S}$.

If σ does not imply the coherence of the final state, we call it a *weak simulation*; in contrast with the strong simulation, the weak one may give false positives, because successful simulation of an upgrade does not guarantee that the actual upgrade will not fail.

A weak simulation σ_2 can be more *accurate* than σ_1, denoted by $\sigma_2 > \sigma_1$, if there exits a coherent state s_i and failing $upgrade(u, op)(s_i) \in Err$ such that:

$$\sigma_2(Mod(s_i), \mathcal{A}(upgrade(u, op))) \in Err \quad \text{but}$$

$$\sigma_1(Mod(s_i), \mathcal{A}(upgrade(u, op))) \in \mathcal{M}$$

Thus, if σ is a simulation, then $\sigma \circ Mod$ is the precondition predicate *pre*, with the desired Property 1.4. Moreover, it is possible to build better approximations of *pre* by iteratively refining the models and the simulation: while obtaining a strong simulation appears unfeasible today, we have already made significant progress refining weak simulations to enhance the overall accuracy.

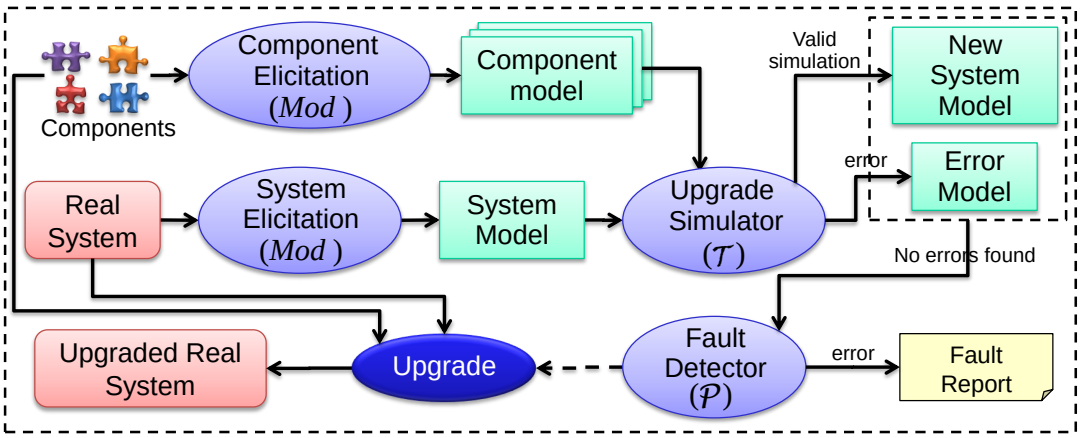


Figure 1.1: Controlled upgrades via model transformations

Figure 1.1 shows the process involving all the actors: *Mod* and σ, composed of $\mathcal{P}$ and $\mathcal{T}$. Real components and real system are elicited by means of the *Component Elicitation* and *System Elicitation*, respectively. These two functions realize *Mod*. The produced models are then taken as input by the *Upgrade Simulator* (that executes $\mathcal{T}$), which produces a *New System Model* or an *Error Model*. If no errors are found the coherence of the produced system model is checked by means of the *Fault Detector* (that evaluates the predicate $\mathcal{P}$). If the system model is coherent the real *Upgrade* can be performed thus producing the *Upgraded Real System*.

In the following we describe the main part composing the approach, namely the model elicitation, see Section 1.2.1, the simulation, see Section 1.2.2, and the fault detector, see Section 1.2.3.

1.2.1 Model elicitation

The elicitation of models representing complex real systems cannot be performed manually, but some automation must be provided. Many approaches have recently addressed the problem of deriving partial behavioral models from implemented systems. For example, [BIPT09] is

an attempt defined in the web-services domain, [UBC09] proposes a synthesis technique that constructs partial behavioural models from safety properties and scenarios, [LMP08] describes a technique to automatically generate behavioral models from object-oriented system execution traces, and [DCDRP+10] presents tools and techniques to automatically derive models from running open source software systems.

1.2.2 Simulation

Smith in [Smi00] defines simulation as "*... the process of designing a model of a real or imagined system and conducting experiments with this model to understand the behavior of the system or to evaluate strategies for its operation.*". An important aspect of the simulation is its validity in terms of fitness for purpose, as emphasized in [PAG+10, Sar86]. Model-driven techniques might help realizing 'trusted' simulations. More specifically, the automation of the construction of the system model on which the simulation is performed is a way to encode the trusting relationship within the elicitation: it is in fact sufficient to largely test the elicitation function for trusting on the built model. Moreover, the elicitation can be executed before each simulation so to help on maintaining the 'trust' [PAG+10] on the models used for the simulation. Finally, model-driven techniques might help in automatically validating the simulator itself. In fact the model produced by the simulator can be compared with the model elicited by the real system by means of a model elicitation technique [DPD11].

1.2.3 Fault detector

The fault detector can be used to check a system model for incoherences. Depending of the nature of the model, different static analysis methods such as model checking [PPS08] or abstract interpretation [CC77] can be applied. A particular class of formal methods are lightweight formal methods, which promise to offer a cost-effective and pragmatic way of improving the quality of software specifications [Jac02]. The EVOSS fault detector is a sort of lightweight formal method. It is fully automatic, focused in the specific domain, and effective. We use OCL[1] instead of using an existent lightweight approach, such as Alloy, since OCL can be directly used to query any Meta-Object Facility (MOF) meta-model and search for model elements denoting fault.

1.3 The case of FOSS distributions

We then apply the general approach to the context of Free and Open Source Software (FOSS) systems that suffer of the problems above discussed [MBC+06]: the community-centric management of such systems is driven not only by the potential difference between current capabilities and the demands of the environment, but also by (i) frequent releases of components [Ray01], and (ii) security issues which require system administrators to immediately react to reduce vulnerability windows [WLC01]. In particular, current upgrade management tools (e.g., the package managers in Linux distributions) are only aware of *some* static dependencies that can influence upgrades and completely ignore relevant dynamic aspects, such as potential failures of configuration scripts that are executed during upgrade deployment. Thus, it is no surprise that an apparently innocuous package upgrade can end up with a *broken* system state [CKK+07].

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

1.4 Structure of the deliverable

This deliverable is structured in 8 chapters:

- Chapter 2 gives an overview of FOSS concepts and highlights limitations of current package management solutions.
- Chapter 3 introduces EVOSS and its composing parts.
- Chapter 4 presents the upgrade simulator component.
- Chapter 5 presents the fault detector component.
- Chapter 6 discusses related works.
- Chapter 7 concludes with final remarks and future research directions.

1.5 Glossary

This section contains a glossary of essential terms which are used throughout this specification.

Distribution A collection of software packages that are designed to be installed on a common software platform. Distributions may come in different flavors, and the set of available software packages generally varies over time. Examples of distributions are Mandriva, Caixa Mágica, Pixart, Fedora or Debian, which all provide software packages for the the GNU/Linux platform (and probably others). The term *distribution* is used to denote both a collection of software packages, such as the *lenny* distribution of Debian, and the entity that produces and publishes such a collection, such as Mandriva, Caixa Mágica or Pixart. The latter are sometimes also referred to as *distributors*.

Still, the notion of distribution is not necessarily bound to FOSS package distributions, other platforms (e.g. Eclipse plugins, LaTeX packages, Perl packages, etc.) have similar distributions, similar problems, and can have their upgrade problems encoded in Common Upgradeability Description Format (CUDF).

Installer The software tool actually responsible for physically installing (or un-installing) a package on a machine. This task particularly consists in unpacking files that come as an archive bundle, installing them on the user machine to persistent memory, probably executing configuration programs specific to that package, and updating the global system information on the user machine. Downloading packages and resolving dependencies between packages are in general beyond the scope of the installer and are what differentiates a meta-installer from an installer. For instance, the installer of the Debian distribution is `dpkg`, while the installer used in the RPM family of distributions is `rpm`.

Meta-installer , also known as a Package Management System. The software tool responsible for organizing a user request to modify the collection of installed packages. This particularly involves determining the secondary actions that are necessary to satisfy a user request to install or de-install packages. To this end, a package system allows the declaration of relations between packages such as dependencies and conflicts. The meta-installer is also responsible for downloading necessary packages. Examples of meta-installers are `apt-get`, `aptitude` and `URPMi`.

Package A bundle of software artifacts that may be installed on a machine as an atomic unit, i.e. packages define the granularity at which software can be added to or removed from machines. A package typically contains an archive of files to be installed on a machine, programs to be executed at various stages of the installation or de-installation of a package, and metadata.

Package status A set of metadata maintained by the installer about packages currently installed on a machine. The package status is used by the installer as a model of the software installed on a machine and kept up to date upon package installation and removal. The kind of metadata stored for each package varies between distributions, but typically comprises package identifiers (usually name and version), human-oriented information such as a description of what the package contains and a formal declaration of the inter-package relationships of a package. Inter-package relationships can usually state package requirements (which packages are needed for a given one to work properly) and conflicts (which packages cannot coexist with a given one).

Package universe, is the collection of packages available through sources known to the meta-installer in addition to those already known by the installer, which are stored in the local package status. Packages belonging to the package universe are not necessarily available on the local machine—while those belonging to the package status usually are—but are accessible in some way, for example via download from remote package repositories.

Upgrade request A request to alter the package status issued by a user (typically the system administrator) using a meta-installer. The expressiveness of the request language varies with the meta-installer, but typically enables requiring the installation of packages which were not previously installed, the removal of currently installed packages, and the upgrade to newer version of packages currently installed.

Upgrade problem The situation in which a user submits an upgrade request, or any abstract representation of such a situation. The representation includes all the information needed to recreate the situation elsewhere, at the very minimum they are: package status, package universe and upgrade request. Note that, in spite of its name, an upgrade problem is not necessarily related to a request to "upgrade" one or more packages to newer versions, but may also be a request to install or remove packages. Both Distribution Upgradeability Description Format (DUDF) and CUDF documents are meant to encode upgrade problems for different purposes.

Chapter 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 $u = \langle c, d \rangle$ [DTZ08] is shown in Figure 2.1.

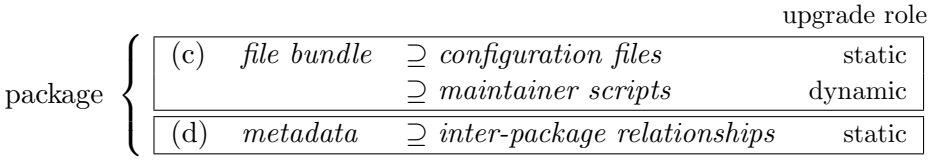


Figure 2.1: Structure of a package

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

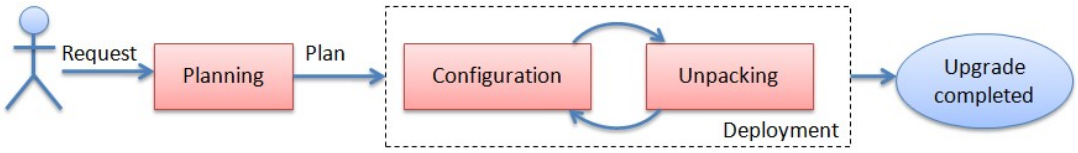


Figure 2.2: The upgrade process: main phases

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.

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) [MBC+06]. 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 2.2 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 [MBC+06]) and choosing the best configuration according to user preferences [TZ09]. 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 2.1).

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 2.1.

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 [DTZ08]. Both experience and previous research show that failures are not hypothetical, but rather the reality of sysadm life [CKK+07]. 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 [XA06]), 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. (i.e., $\neg Coher(s_f)$ in Section 1). 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 2.1 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 *fault detector* component, which is able to deal with undetected failures.

Chapter 3

Evoss: EVolution of free and Open Source Software

To improve the failure prediction of FOSS system upgrades, we propose a model-driven engineering (MDE) approach [Bó5] called Evoss [DCDRP+10] 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 3.5.

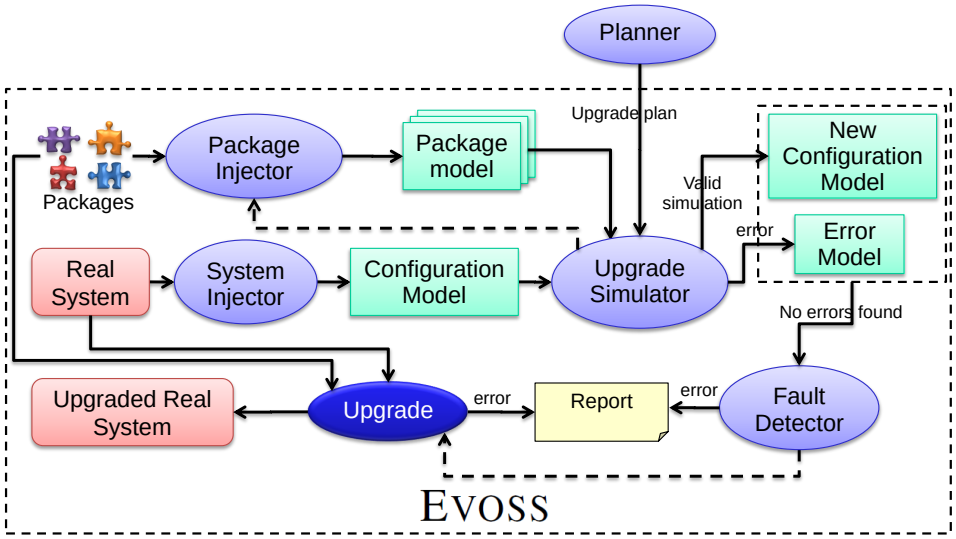


Figure 3.1: Overview of the Evoss approach

An overview of Evoss is sketched in Figure 3.1. The simulation of a system upgrade is performed by the *Upgrade Simulator* [DPD11] 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*

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 3.2: Modeling packages

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 *Fault Detector* component [DP11]. When *Fault 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.

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 3.2.

The modeling constructs which can be used to define system and package models are defined in the *system configuration* and *package* metamodels, respectively [DPPZ09]. 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 3.3 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.

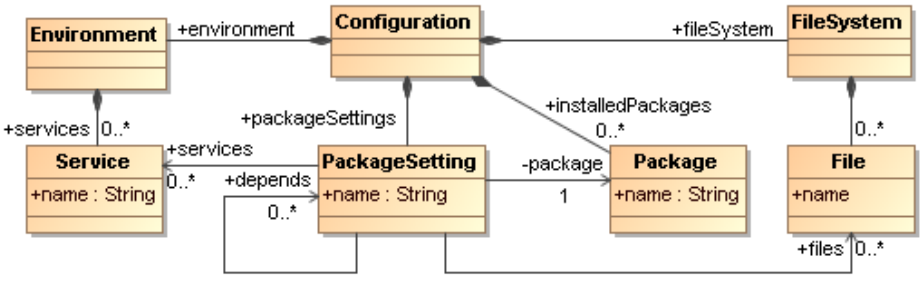


Figure 3.3: Fragment of the Configuration metamodel

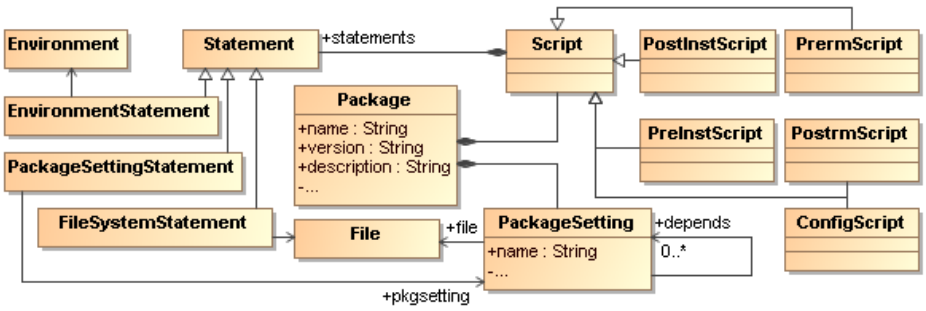


Figure 3.4: Fragment of the Package metamodel

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 that allows us to capture undetected failures, like the one that would arise if modifying the scripts of Example 2.1.

Package metamodel The metamodel shown in Figure 3.4 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 3.2 describes the upgrade simulator component, Section 3.3 describes the fault 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 [DPP10] that is implemented by using the Eclipse Modeling Framework (EMF)[1]. As shown in Figure 3.5, this architecture has a layered

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

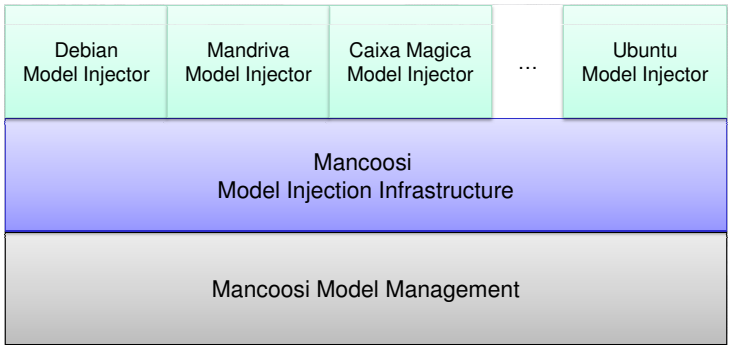


Figure 3.5: The Evoss model injection architecture

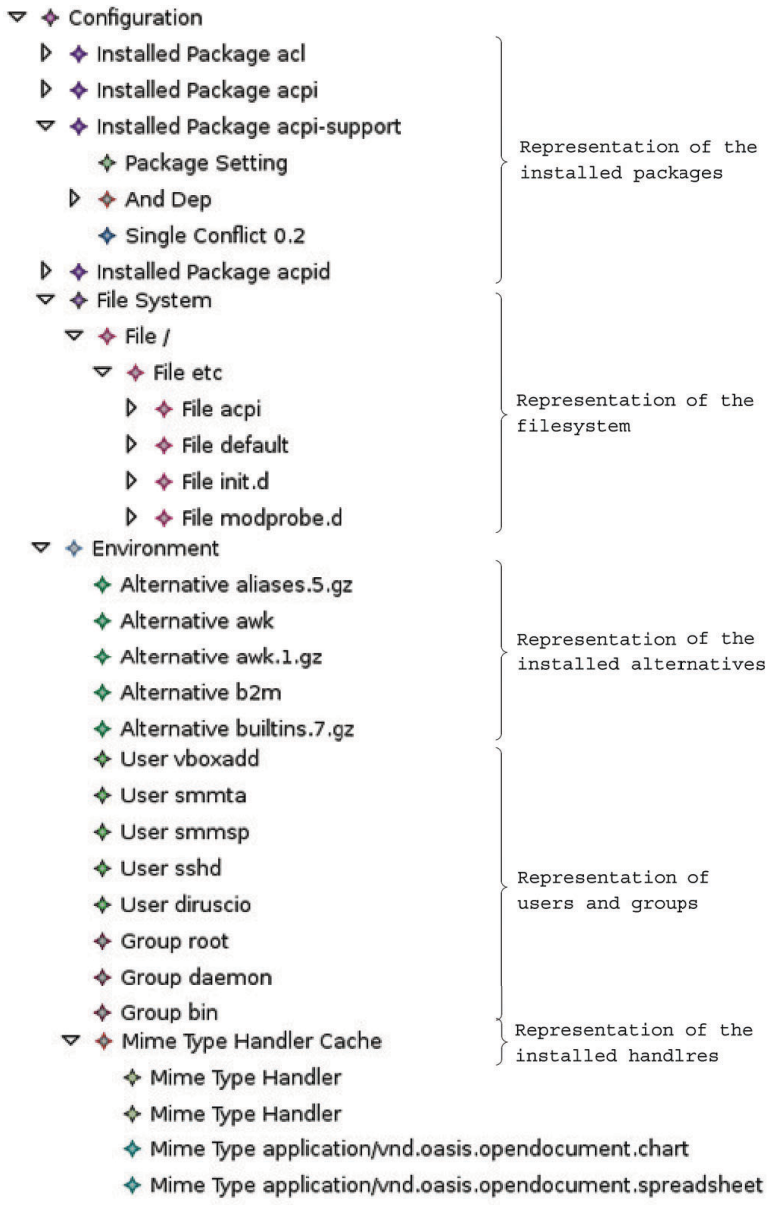


Figure 3.6: Configuration model: an example

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 the infrastructure necessary to manipulate models conforming to the Configuration and Package metamodels shown in Figure 3.3 and in Figure 3.4. 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 3.5.

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 3.6 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.

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 3.6) and both are associated with the corresponding files which store their configurations. Such a fine-grained, installation-specific dependency is not currently expressible using package metadata only.

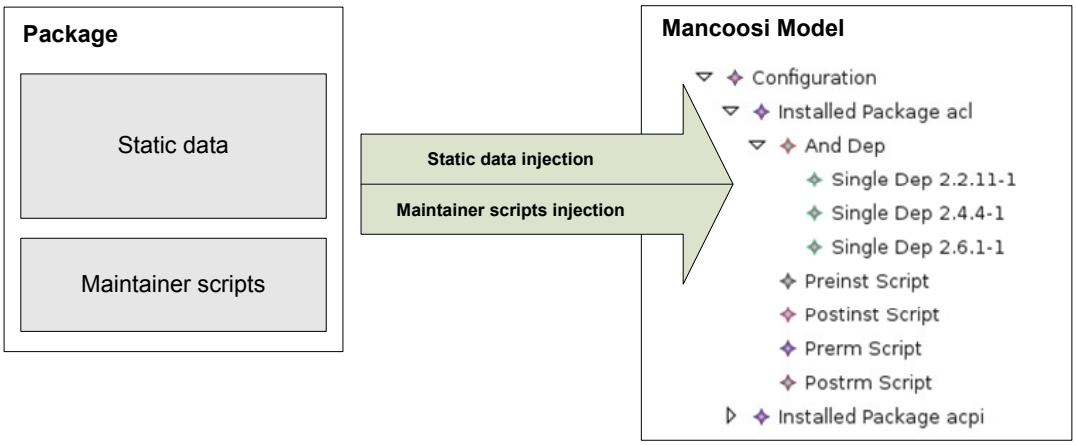


Figure 3.7: Overview of the package injection procedure

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 3.7, 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 [JJJ08] 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.

The *Package Model* provides information about the package involved in the upgrade, including maintainer scripts behaviour. Figure 3.8 shows a fragment of the package `swiprolog-5.7.59`.

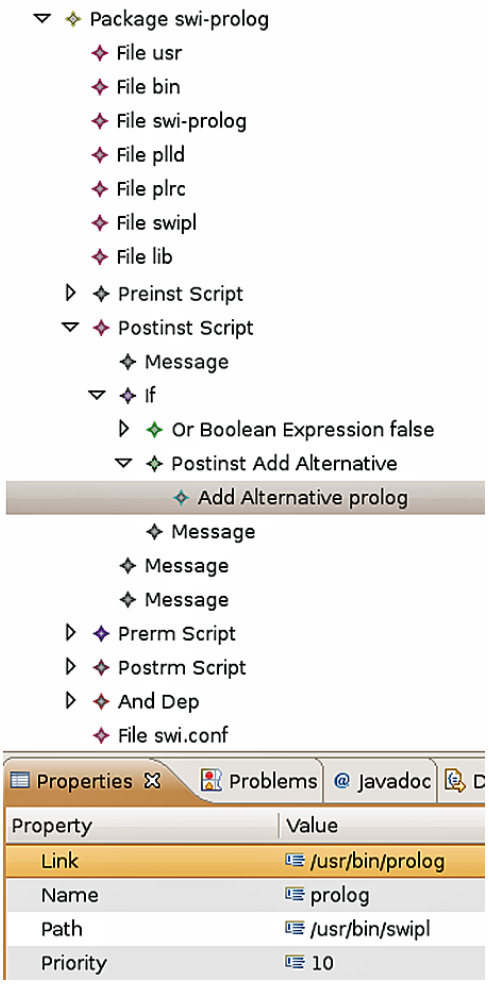


Figure 3.8: Model of the sample package `swiprolog-5.7.59`

The model represents the files contained in the package, the dependencies with other packages and the maintainer scripts which are executed during the upgrade. In EVOSS, maintainer scripts can be expressed by means of a DSL [DCDRP+10] that mainly provides macros representing recurring script fragments. Moreover, the DSL contains limited control flow operations and a tagging mechanism that allows us to specify the behavior of script parts that cannot be completely specified with DSL macros. This way, script authors (usually package maintainers) can specify how such parts affect the configuration model and enable their simulation. The limited expressive power of the DSL is the price to be paid to have the DSL amenable to automated analysis.

The postinst script of the package `swiprolog-5.7.59` consists of four DSL statements (see Figure 3.8). The statement that can affect the system configuration during the package installation is the `if` statement. Its `then` block contains the `addAlternative` statement that, once executed, creates a new alternative called `prolog`, which points to the executable `/usr/bin/swipl`. In particular, the alternative system used by FOSS aims to solve problems which can rise when several programs fulfilling the same or similar functions are installed at the same time. For example, a system may have several prolog engines at once. This permits users to choose the engines to execute by means of the alternatives system. A generic name in the filesystem is

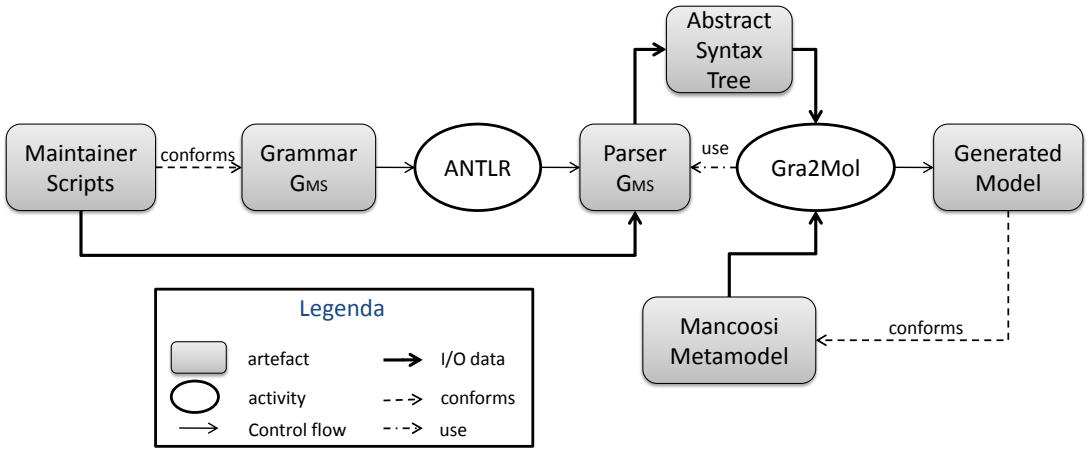


Figure 3.9: Maintainer scripts injection

shared by all files providing the interchangeable functionality. For example, if both `SWI-Prolog`[2] and `AI::Prolog`[3] are installed on the system, the alternatives system can cause the generic name `/usr/bin/prolog` (also named master link) to refer to `/usr/bin/swipl` (also named slave link) by default. The system administrator can override this by changing the reference of the generic name to link to `/usr/bin/aiprolog`.

The maintainer scripts injection process is depicted in Figure 3.9: G_{MS} is the grammar we defined for parsing the maintainer scripts. By means of ANTLR[4] 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 Gra2MoL transformations which query it and generate target models. Further details about the whole injection process can be found at [DTPP09].

3.2 Upgrade Simulator

The simulation of system upgrades is performed by the *Upgrade Simulator* component shown in Figure 3.1. 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 3.10. 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.

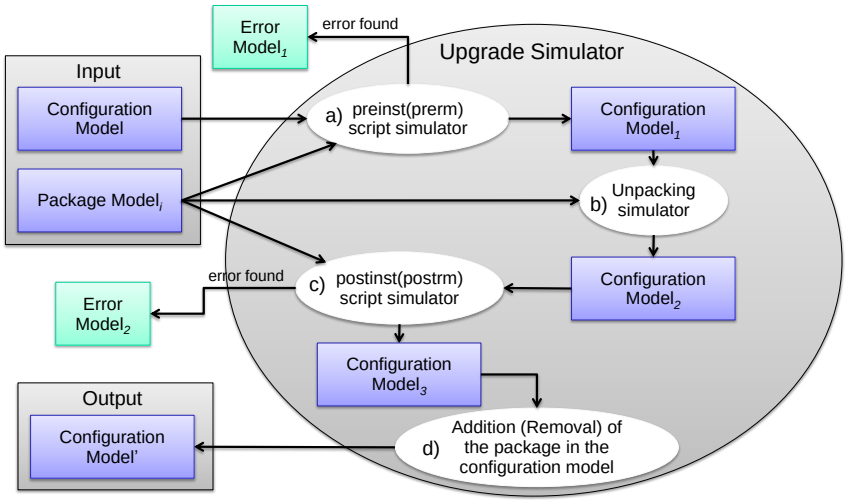


Figure 3.10: Upgrade simulator

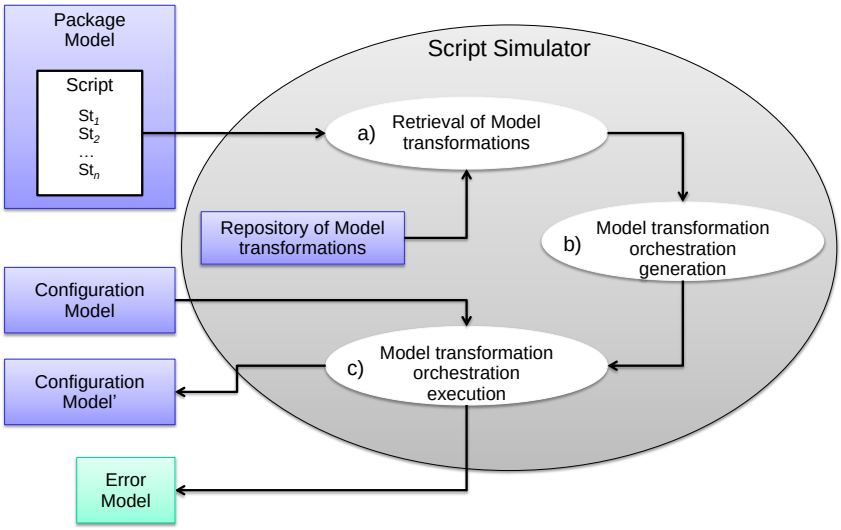


Figure 3.11: Script simulator

The most delicate part of the overall process is the simulation of the scripts which are executed during the upgrade. Figure 3.11 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, 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.

[2] `http://www.swi-prolog.org/`
[3] `http://cpan.uwinnipeg.ca/dist/AI-Prolog`
[4] `http://www.antlr.org`

3.3 Fault detector

Given the current configuration model, the system upgrade is simulated by taking into account the packages that have to be upgraded. A *fault 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 fault detector. In particular, for each detectable classes of faults, a corresponding OCL[5] expression is defined and used to query models and search for model elements denoting faults.

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 3.1: 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).

Listing 3.1: Fragment of the OCL query to detect missing mime type handlers

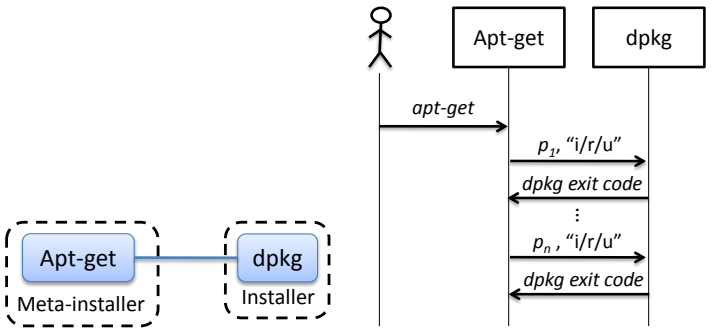
```
1 MM!MimeTypeHandler.allInstancesFrom('IN')
2       ->select(e |
3                   e.handler.oclIsUndefined()
4               )->size()
```

The fault 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

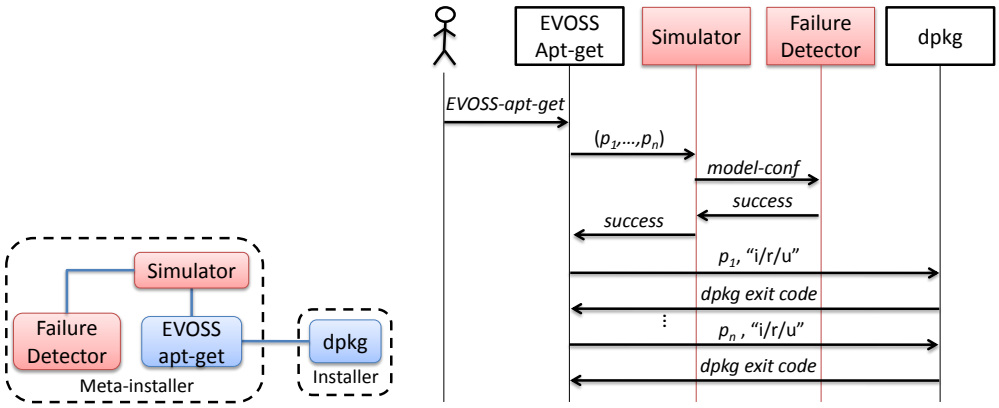
The integration of Evoss with legacy environments is realized by suitably extending existing meta-installers. Figure 3.12 shows both the static and dynamic views of the popular meta-installer *apt-get*. Its architecture is very similar to other meta-installer architectures. As shown

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



(a) Static view (b) Dynamic view

Figure 3.12: The *apt-get* meta-istaller: current solution



(a) Static view (b) Dynamic view

Figure 3.13: Meta-istallers enhanced by EVOSS

in Figure 3.12(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 3.13 shows the enhancement of existing meta-installers by means of the *Simulator* and the *Fault Detector* component provided by EVOSS. The upgrade request is sent to the *Simulator* component before performing the upgrade on the real system. Figure 3.13(b) shows the case in which both the *Simulator* and the *Fault 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 *Fault Detector* components detect errors, then the upgrade is stopped and the user is informed about the encountered problems.

3.5 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 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 predicting 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[6], a very large distribution [GRM+09] based on the `dpkg` installer, and Fedora[7], based on the `RPM` installer.

This analysis allowed both the identification of recurring templates in maintainer scripts and

[6] `http://www.debian.org`
[7] `http://fedoraproject.org`

the collection of them in a limited number of clusters. The adopted procedure and results can be found in [DRPPZ09] 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[8] 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 [DPPZ09]. Each one of those templates contains statements that are executed as a whole. Listing 3.2 shows a template example consisting of statements which get executed after the removal of GNOME[9] components which ship GNOME configuration schemata.

Listing 3.2: 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 3.4. The `Statement` metaclass represents an abstraction of the commands that can be executed 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 3.14: 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 [DTPP09].

[8] 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`.

[9] 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 3.14: Fragment of the DSL grammar

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 3.14), 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 [CDP07] 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:

```
1 if [ -e /etc/apache2/apache2.conf ] ; then
2     a2enmod php5 >/dev/null || true
3     reload_apache
4 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:

```
1 #<%
2 addPackageSetting_dependences(apache2,php5);
3 addEnvironment_runningServices(env,apache2);
4 #%if [ -e /etc/apache2/apache2.conf ] ; then
5 #%     a2enmod php5 >/dev/null || true
6 #%     reload_apache
7 #%fi
```

```
8 #%>
```

In Example 3.5, 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[10]). 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 3.5).

Example 3 The semantics of the statement `rm_alternative` is shown in Listing 3.3: 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 3.3: 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 }
```

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

Chapter 4

Upgrade Simulator

In this chapter we look into the simulator of Evoss by presenting its structure, its constituting elements, its implementation and by discussing the kind of errors the simulator is able to predict before they affect the real system. The architecture of the simulator is distribution independent so that it can be easily instantiated to the specific distribution. The simulator is based on model-driven techniques and makes use of a model-based description of the system to be upgraded. The simulator is able to simulate also pre and post installation scripts that equip each distribution package. This is possible thanks to a domain specific language (DSL), which has been conceived within Evoss to specify maintainer scripts behavior [DTPP09]. In particular, the language includes a set of high level clauses with a well-defined transformational semantics expressed in terms of system configuration modifications: each system configuration is given as a model and the script behavior is represented by corresponding model transformations.

Real experiences show how the simulator works in practice by highlighting its efficacy, i.e., how it improves the state of the art of package managers, and its performance while integrated in real Linux distribution installations.

This chapter is organized as follows. Section 4.1 provides an overview of the upgrade simulator. Section 4.2 presents three upgrade scenarios that the simulator is able to perform. Section 4.3 presents the simulation of maintainer scripts. Section 4.4 describes how the simulator has been implemented by using model-driven technologies. The simulator can be integrated with current Linux distributions and this is presented in Section 4.5. Section 4.6 presents a large experimentation which has the aim of validating the approach and of showing its applicability in practice.

4.1 Overview

The *Upgrade Simulator* is part of the Evoss approach and its integration with the other components of Evoss is shown in Figure 3.1. This section has the aim of providing an overview of the upgrade simulator.

As shown in Figure 4.1 the upgrade simulator gets as input both an upgrade plan, which is produced by a planner and consists of a sequence of packages with associated the operation to be performed (i.e., installation and removal of a package), and a configuration model. The upgrade simulator invokes the Evoss package injector that gets as input a compiled package

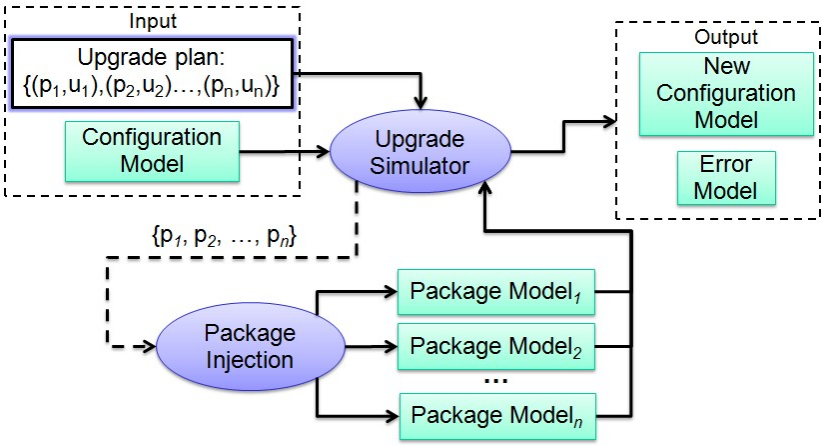


Figure 4.1: Upgrade simulator within EVOSS

and automatically produces a model representing both static aspects of the package, i.e., the information contained into the package metadata, and dynamic aspects, i.e., maintainer scripts. The models, one for each package, that are produced by the EVOSS package injector are provided to the upgrade simulator. Then the upgrade simulator performs the upgrade simulation and produces as output a new configuration model or an error model if some errors are discovered.

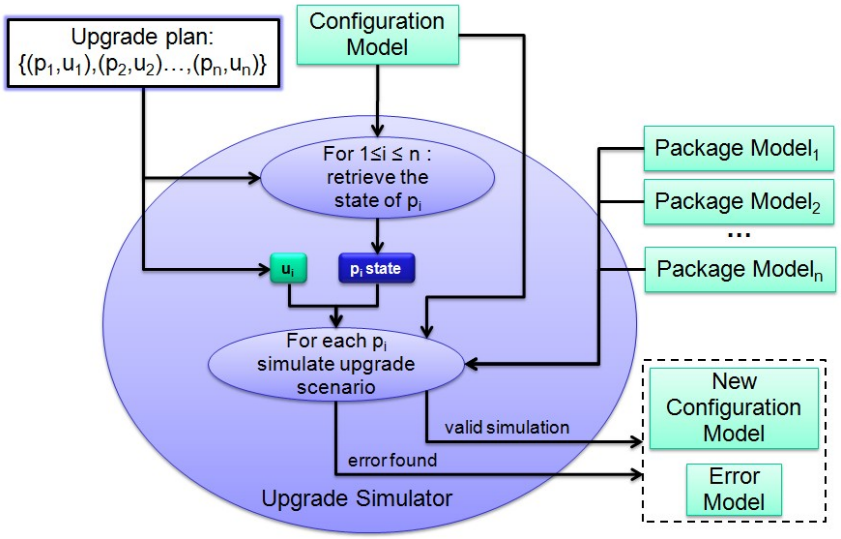


Figure 4.2: Upgrade simulator

As shown in Figure 4.2, the upgrade simulator, given an installation plan $\{(p_1, u_1),(p_2, u_2),\cdots,(p_n, u_n)\}$, retrieves the state of each package p_i, $1 \leq i \leq n$. It is important to note that the installation plan provides an ordered set of pairs, composed of a package and the operation to be performed; then the upgrade simulator simulates the configuration upgrade by respecting the installation plan order. Simulating the upgrade plan corresponds to simulating different upgrade scenarios according to the states that the considered packages can have in the configuration model. In particular, the possible states of a package, which are relevant for the simulation purposes, are[1]:

[1] According to the Debian policy `http://www.debian.org/doc/debian-policy` and the manpage of `dpkg`

- `Installed`: the package is installed.
- `Not installed`: the package is not installed and the configuration files are not present in the system.
- `Half-installed`: when installing or removing the package some error has been experienced and then the package is left in a half-installed state. This state can also have a `Reinst required` flag meaning that the package needs to be reinstalled even if we want to remove it.
- `Config-files`: when removing a package that was previously installed, if everything goes well, the package will be left in the `Config-files` state since the configuration files of the package have not been deleted. The `purge` operation must be performed to delete also the configuration files and then to reach the `Not installed` state.
- `Halfconfigured`: indicates that the package has been unpacked, but its configuration scripts have failed.
- `Unpacked`: The package is unpacked, but not configured.

The output of the selected simulation scenario is a *New Configuration Model* and an *Error Model* if some errors are found while simulating the scenario. More precisely, a simulation is valid if no failures are experienced during the simulation journey. Note that, even when the simulation is not valid a new configuration model is produced; this might help in identifying the error (the exact script and statement that raised it) and understanding it.

4.2 Upgrade scenarios

In this section we present three upgrade scenarios that the simulator is able to perform:

- *Install* - this scenario can be performed in six different states:
  - `Not installed` state described in Section 4.2.1;
  - `Config-Files` state described in Section 4.2.2;
  - `Half-installed` or `Half-configured` state described in Section 4.2.3;
  - `Installed V1` state described in Section 4.2.4. This is a particular install scenario in which we perform the installation of a package already installed (version `V1`).
- *Remove* - this scenario can be performed in three different states:
  - `Installed` state described in Section 4.2.5;
  - `Half-installed` or `Half-configured` states described in Section 4.2.6).
- *Purge* - this scenario can be performed in four different states:
  - `Config-Files` state described in Section 4.2.7;
  - `Half-installed` or `Half-configured` states described in Section 4.2.8;
  - `Installed` state described in Section 4.2.9.

`http://man.cx/dpkg#sec5`

4.2.1 Install performed in a `Not installed` state:

Figure 4.3 shows the operations that are performed when an installation has to be performed and the considered package is in the `Not installed` state. The `preinst` maintainer script of the package is invoked with `install` as parameter. If the script is simulated with no errors, the unpacking of the files is simulated. This simulation consists of adding in the configuration model the representation of the files contained in the package. Then the `postinst` script is invoked with `configure` as parameter. If the simulation of this script fails then an error model is produced and the package transits in the `Failed config` state. In case of the simulation of the `preinst` script fails then the simulation of the `postrm` script with parameter `abort-install` is invoked with the aim of removing the effect of the failed `preinst` script invoked before. If the simulation of the `postrm` script successfully terminates then the `Not installed` state is reached and an error model, which describes what caused the failure of the `preinst` script, is produced. If the simulation of the `postrm` script fails a `Half installed` state is reached with associated the `Reinst required` flag meaning that the package needs to be reinstalled even if we want to remove it.

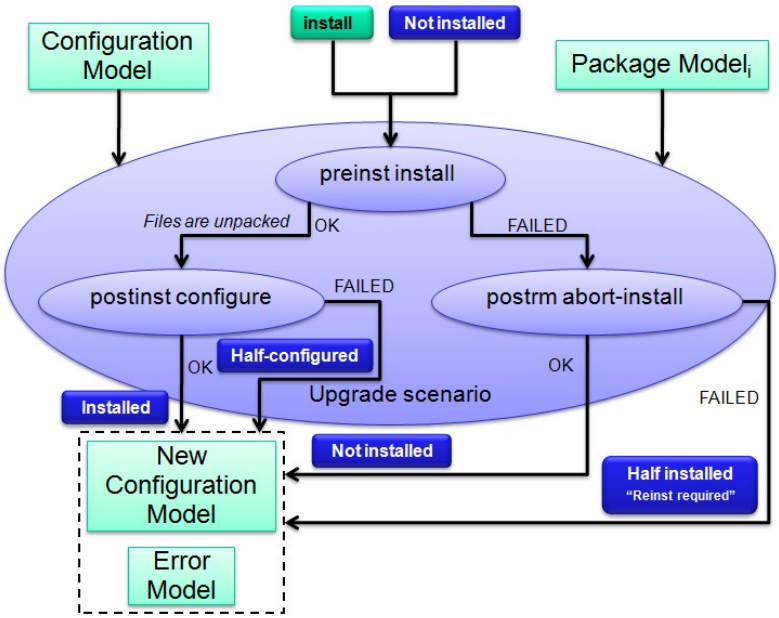


Figure 4.3: Install performed in a `Not installed` state

4.2.2 Install performed in a `Config-Files` state:

Figure 4.4 shows the operations that are performed when an installation has to be performed and the considered package is in the `Config-Files` state. This scenario is very similar to the previous one: the involved scripts, which are obviously of the new version of the script (version `V2`), are invoked with `V1` as parameter, where `V1` is the version of the package previously installed and then removed. This makes the scripts aware of the existence of the configuration files of the package version `V1`.

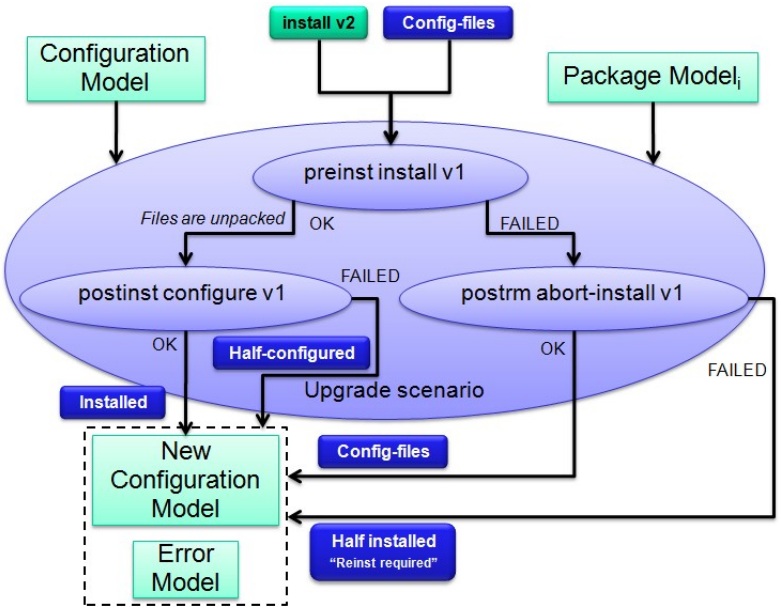


Figure 4.4: Install performed in a `Config-files` state

4.2.3 Install performed in a `Half-installed` or `Half-configured` state:

The scenario of installation of a package that is in a `Half-installed` or `Half-configured` state is somehow in between the scenario described in Section 4.2.1 and the one described in 4.2.2. In fact, we cannot exclude that configuration files of the package are present in the system, but on the other side we are not sure that these files are present. Moreover, configuration files might be of different package versions.

4.2.4 Install performed in an `Installed` state (`V1`):

This concerns the upgrade of an installed package. In particular the existing version `V1` is updated with the new version `V2`. The upgrade of an installed package is more complicated since involves both the scripts of the installed version and those of the version to be installed. The overall scenario is shown in Figure 4.5 that describes a particular installation in which a previous version of the package to be installed in already present in the system.

The basic actions that are performed are the invocation of maintainer scripts in the following order:

1. Execute the `prerm` script of the package `V1` with parameters `upgrade` and `V2`;
2. Execute the `preinst` script of the package `V2` with parameters `upgrade` and `V1`;
3. Files are unpacked;
4. Execute the `postrm` script of the package `V1` with parameters `upgrade` and `V2`;
5. Execute the `postinst` script of the package `V2` with parameters `upgrade` and `V1`.

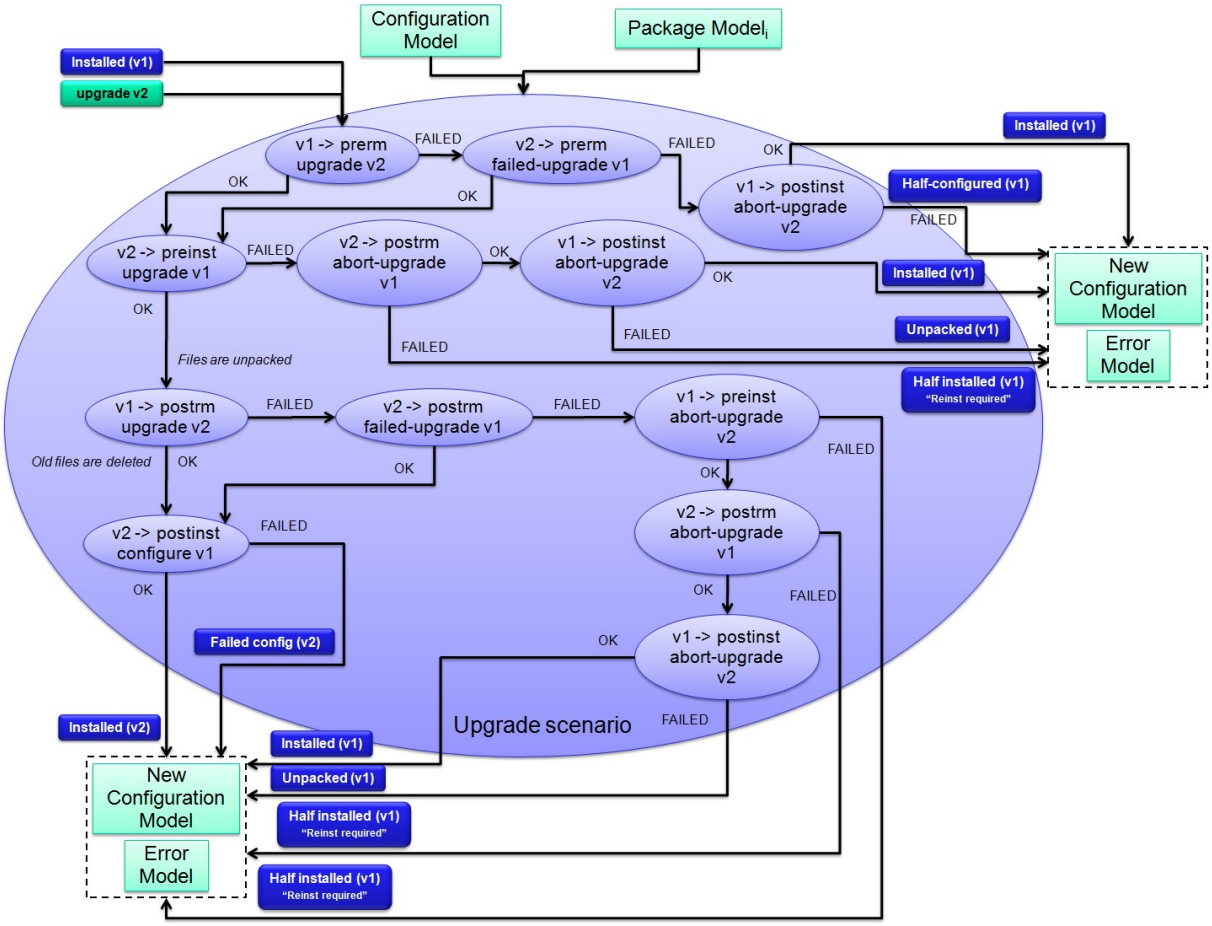


Figure 4.5: Install of a package performed in an `Installed` state

4.2.5 Remove performed in an `Installed` state:

Figure 4.6 shows the operations that are performed when a package in the `Installed` state has to be removed. The `prerm` script of the package is invoked with `remove` as parameter. If the script is simulated with no errors, files associated to the package, but not configuration files, are removed. Then the `postrm` script is invoked with `remove` as parameter. The reached state is `Config-files` meaning that configuration files of the package are still present on the system, i.e., they are not removed. If the simulation of this script fails then an error model is produced and the package transits to the `Half-installed` state. In case of the simulation of the `prerm` script fails then the simulation of the `postinst` script with parameter `abort-remove` is invoked with the aim of removing the effect of the `prerm` script invoked before and failed. If the simulation of the `postinst` script successfully terminates then the `Installed` state is reached and an error model, which describes the motivations of the failure of the `prerm` script, is produced. If the simulation of the `postinst` script fails a `Half installed` state is reached.

4.2.6 Remove performed in a `Half-installed` or `Half-configured` state:

Removing a package that is in a `Half-installed` or `Half-configured` state consists of reinstalling and then of removing the package.

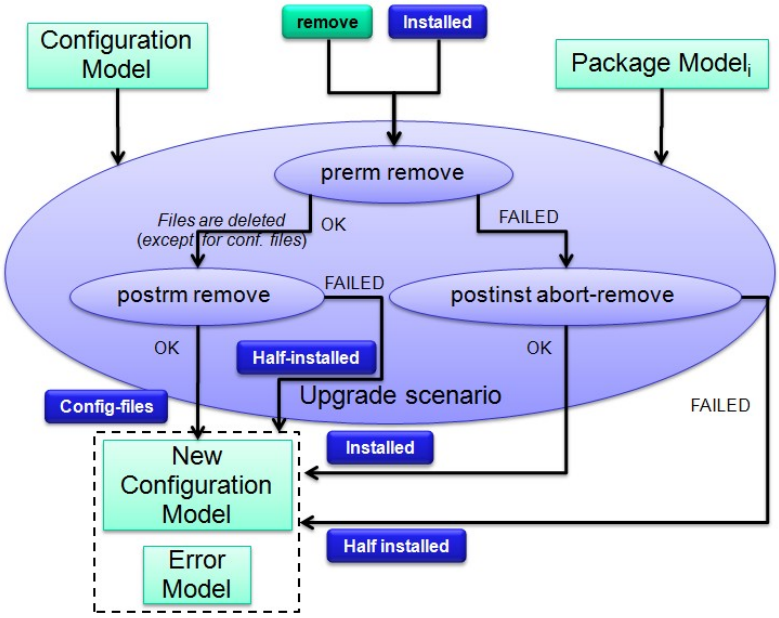


Figure 4.6: Remove performed in an `Installed` state

4.2.7 Purge performed in a `Config-Files` state:

Figure 4.7 shows the purging of a package performed in a `Config-Files` state. A `Config-Files` state may be reached when an installed package has been removed and, as explained in Section 4.2.5, the configuration files of the package are not removed. Therefore, the simulation of the `postrm` script is performed with `purge` as parameter. If the script terminates without errors, the files are removed and the state of the package is set to `Not-installed`. If the script fails, then an error model is produced and the state of the package does not change, i.e., it remains in the `Config-Files` state.

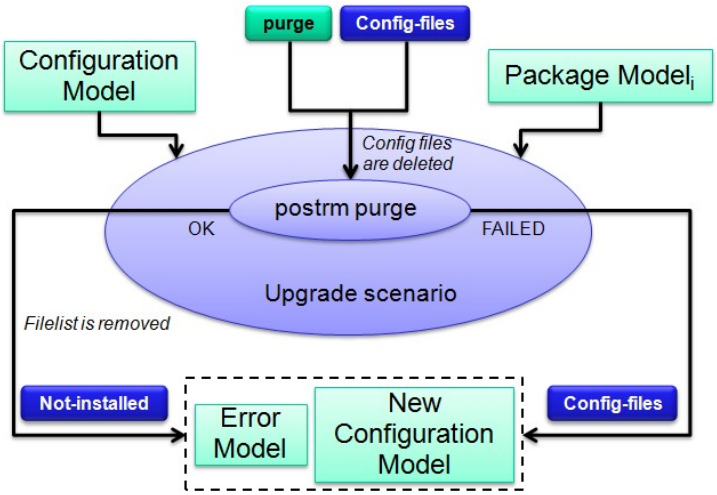


Figure 4.7: Purge performed in a `Config-files` state

4.2.8 Purge performed in a `Half-installed` or `Half-configured` state:

Purging a package that is in a `Half-installed` or `Half-configured` state consists of reinstalling and then of purging the package.

4.2.9 Purge performed in an `Installed` state:

Figure 4.8 shows the removal and purging scenario that consists of the union of the removal and purging scenarios.

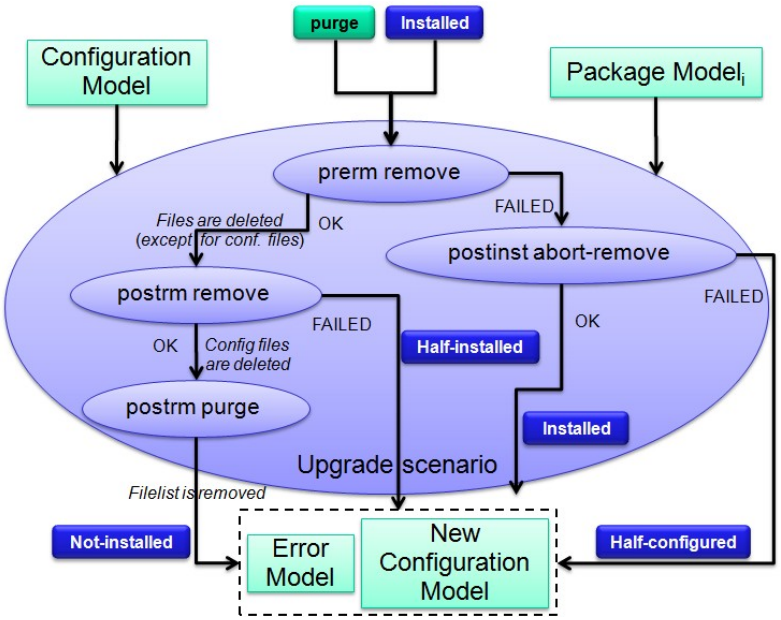


Figure 4.8: Purge performed in an `Installed` state

4.3 Maintainer scripts simulation

As shown in each scenario described in Section 4.2, an important and relevant part of the configuration upgrade simulation is the simulation of maintainer scripts. As described before and shown in Figure 4.1, for each package involved in the upgrade, the package injector retrieves the maintainer scripts associated to the package and transforms them in DSL scripts. Then the idea is to execute the model transformation that is associated to the DSL statements, in order to understand how the execution of the statements composing the script affects a source configuration model.

Figure 4.9 shows an overview of the maintainer script simulator. The script simulator performs three subsequent activities. Given a package model that contains some maintainer scripts expressed in the DSL, it selects the script to be simulated. Let the script be composed of n statements $st_1, st_2, \ldots, st_n$. The first activity, 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

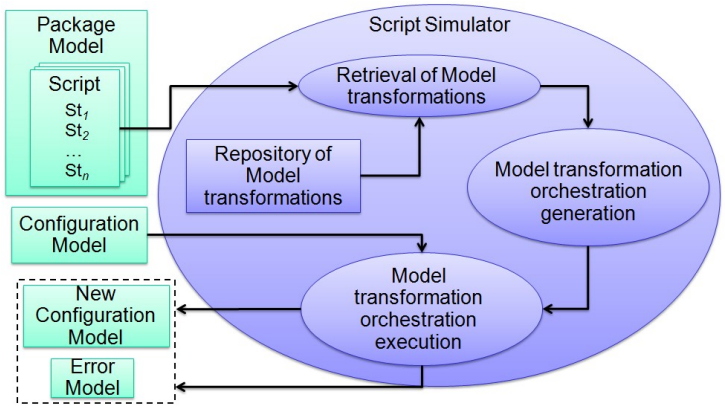


Figure 4.9: Maintainer script simulator

provide the (operational) semantics of the script statements. More precisely, the model transformations define how a source configuration changes when DSL statements are executed. The second activity, namely *Model transformation orchestration generation*, properly chains these model transformations. Finally, the last activity, namely *Model transformation orchestration execution*, executes the chain of transformations on the source configuration model and, if no error is encountered at any step, generates a new configuration model. In case of error, an error model is produced.

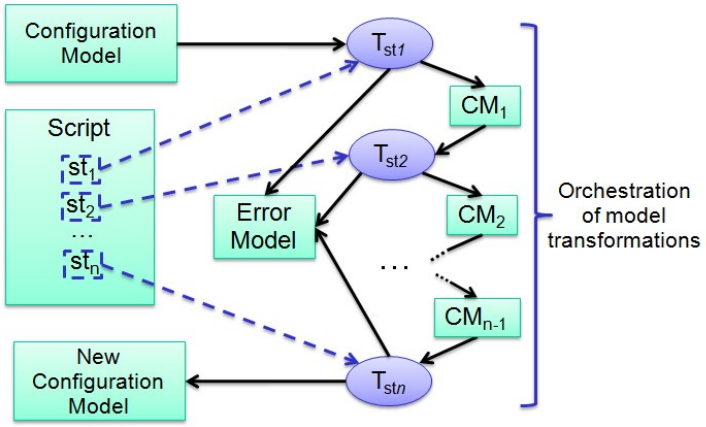


Figure 4.10: Orchestration of maintainer script statements

Figure 4.10 details the orchestration of the n statements st_1, st_2, $\cdots$, st_n composing the considered maintainer script. As previously said, the maintainer script is written in the EVOSS DSL and then each statement has its semantics expressed in terms of model transformations. Then, for each i, $1 \leq i \leq n$, let T_{st_i} be the transformation associated to the statement st_i. The transformation T_{st_1} gets as input the initial configuration model and produces as output the configuration CM_1 if no errors are found, otherwise an error model is produced, the simulation of the script is stopped, and the control is passed to the upgrade scenario that invoked the script. The execution of each statement is suitably orchestrated and finally (if no errors are found) after the simulation of T_{st_n} with input CM_{n-1} produces *New Configuration Model*. It is important to note that even in case of errors, the maintainer scripts simulation returns, together with an error model, the last configuration model that has been produced during the simulation.

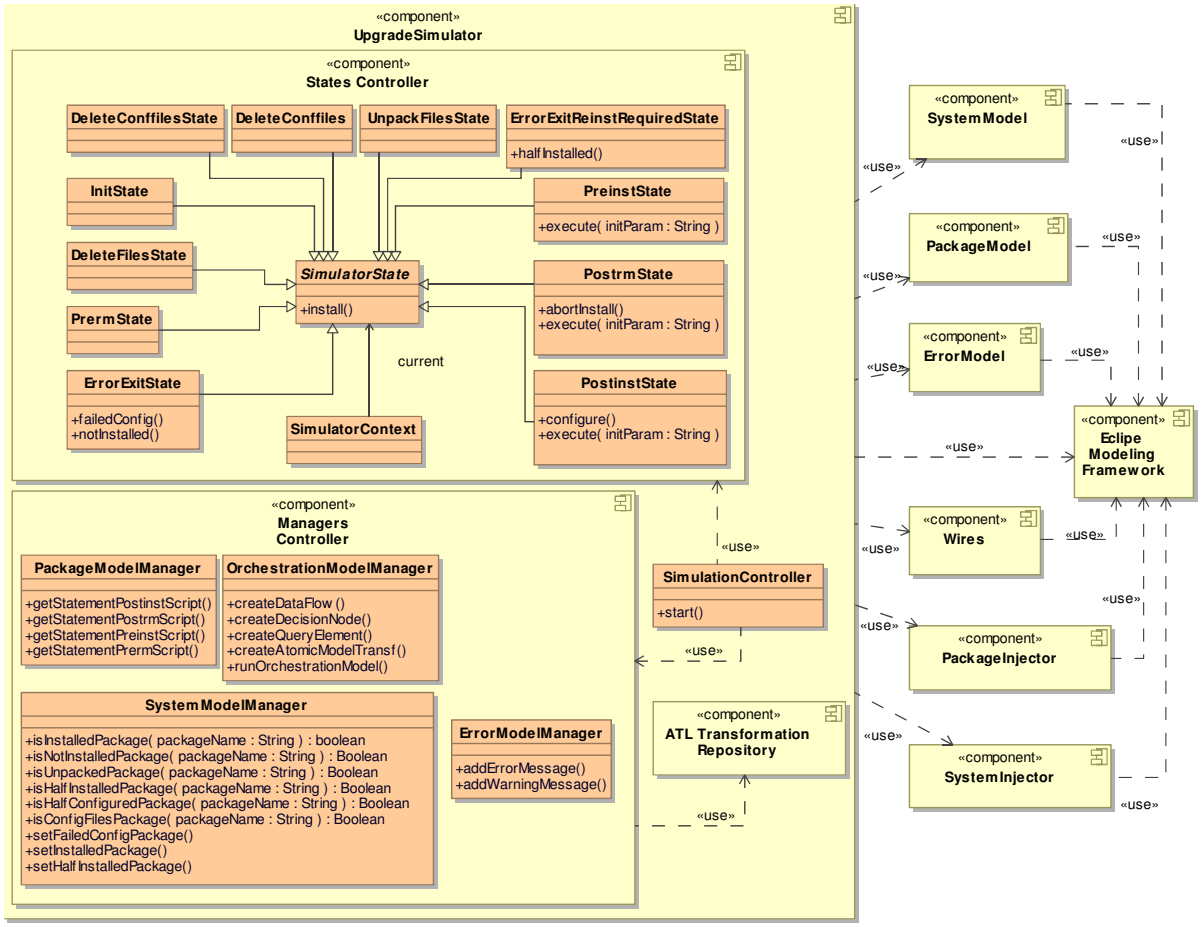


Figure 4.11: Simulator architecture

4.4 Implementing the Upgrade Simulator

In this section we present an implementation of the Upgrade Simulator discussed in the previous section. The simulator has been conceived by using different model-driven techniques and tools based on the Eclipse Modeling Framework (EMF) [BSM+03]. The implementation of the simulator together with the implementation of the whole EVOSS approach is freely available at `http://evoss.di.univaq.it`. Section 4.4.1 presents the architecture of the simulator, whereas Section 4.4.2 describes more insights related to the execution of the simulator on concrete scenarios.

4.4.1 Overview

The architecture of the upgrade simulator is shown in Figure 4.11. The `Simulator` component mainly consists of three sub-components named `StatesController`, `ManagersController`, and `ATLTransformationRepository`.

The `StatesController` component provides elements to maintain the state of the simulator during the simulation of a given upgrade scenario. In this respect, the `StatesController` implements the *State design pattern* [GHJV95], which provides a solution to the problem of having

different behaviors each depending on the state. According to such design pattern, the abstract class `SimulatorState` is defined and the different states are specified as derived classes of the simulator state base class. Such derived classes provide state-specific behaviors for the methods defined in the base class. In other words, `StatesController` permits to maintain finite state machines each representing the execution of an upgrade scenario. For instance, the class `InitState` represents the initial state of any upgrade scenario being simulated. Therefore, the implementation of the `install` method in `InitState` has a different behaviour of that, for instance, of `PostinstState`. In fact, as expected, it is not possible to execute the `install` transition in the `PostinstState`, which, on the contrary, can accept the execution of methods like `configure` to configure a package just installed. The current state of the finite state machine is maintained by the `SimulatorContext` class by means of the `current` relation.

The `ManagerController` component consists of managers able to interact with the external components devoted to the management of all the models which are manipulated during the simulation. For instance, the `SystemModelManager` is an intermediate layer to interact with the `SystemModel` component, which provides user with facilities to modify and query configuration models. For instance, the method `isInstalledPackage` in `SystemModelManager` checks if a given package is installed or not according to the information available in the considered system configuration model. The `ErrorModelManager` is an entry point to the `ErrorModel` component, which is used by the simulator to create error models in case of simulation failures. The models of the packages that are considered in the upgrade plans are manipulated by means of the `PackageModelManager` which has methods to interact with the `PackageModel` component. For instance, given a package model, the execution of the method `getStatementPostInstScript` of `PackageModelManager` returns the statements that should be executed after any package installation simulation.

It is worth mentioning that the `SystemModel`, `ErrorModel`, and `PackageModel` components have been developed starting from the specification of three different Ecore models. Then, by using the Java code generation facilities provided by EMF, starting from such Ecore models three corresponding Java components have been generated. Such components provide the means to manage in a programmatic way models conforming to the defined Ecore models.

The simulator is able to simulate also the maintainer scripts which are executed during the upgrades. In this respect, the simulator is able to create and execute the chain of model transformations each representing the operational semantics of a single statement of the script to be simulated. The upgrade simulator contains the `ATLTransformationRepository` that provides all the model transformations that have to be executed during the upgrade scenarios. Such transformations are implemented in ATL, a QVT compliant language that is part of the AMMA platform [JABK08]. Listing 4.1 contains a fragment of the model transformation that defines the semantics of the `addAlternative` statement of the Evoss DSL used in the maintainer script of the package in Figure 3.8. Such a transformation consists of a module specification containing a header section (line 1), transformation rules (lines 8-48) and a number of helpers (lines 3-6), which are used to navigate models and to define complex calculations on them. In particular, the header specifies the source models, the corresponding metamodels, and the target ones. According to line 1 of Listing 4.1, the `addAlternative` transformation generates a target configuration model (and possibly an error model) starting from (i) the source configuration model, and (ii) the package model representing the package that is involved in the upgrade scenario to be simulated.

Helpers and rules are the constructs used to specify the transformation behaviour. The transformation in Listing 4.1 consists of a set of rules each devoted to copying elements of a given

type (e.g., `Configuration` and `Environment`) from the source configuration model to the target one. For instance, in the fragment shown in Listing 4.1 the `Configuration` rule is able to copy the configuration maintained in the source configuration model to the target one. As outlined in the previous section, a configuration consists of many elements like the installed packages, the configuration files, and the environment. For each of them a corresponding transformation rule is provided, like the rule `Environment` (see lines 23-48) that copies the environment data from the source configuration model to the target one.

Besides such a copy operation, the rule specifies the key semantics of `addAlternative` in the action block specified in lines 36-46. In particular, if the location of the alternative being added (e.g., `/usr/bin/swiprolog` in Figure 3.8) does not exist in the source configuration model, an error model is created to notice the user about the occurred failure (see line 45). If such a location is available and the alternative system is managing an alternative named as the value of the `nameAlt` variable (e.g., `prolog` in Figure 3.8) then the new alternative is added (see line 41). In case of the sample package in Figure 3.8 the new alternative `/usr/bin/swiprolog` will be added to the alternative named `prolog`.

```
1  create OUT : OUTConfiguration, OUTERR: MMError  from IN : INConfiguration, INPACK :
       ↪INPackage, INPOS : INCurrentPosition;
2
3  helper def : existsAlternative(n:String) : Boolean = ...;
4  helper def : getAlternative(n:String) : INConfiguration!Alternative = ...;
5  helper def : existsFile(n:String) : Boolean = INConfiguration!File.allInstancesFrom('IN'
       ↪)->collect(e | e.location)->exists(e | e = n);
6  helper def : getFile(n:String) : INConfiguration!File = INConfiguration!File.
       ↪allInstancesFrom('IN')->select(e | e.location = n)->first();
7  ...
8  rule Configuration {
9      from s : INConfiguration!Configuration
10     to t : OUTConfiguration!Configuration (
11         name <- s.name,
12         creationTime <- s.creationTime,
13         systemType <- s.systemType,
14         installedPackages <- s.installedPackages,
15         fileSystem <- s.fileSystem,
16         environment <- s.environment,
17         ...)
18   do {
19     ('Execution:␣add_alternative_deb.atl').println();
20   }
21 }
22 ...
23 rule Environment {
24     from s : INConfiguration!Environment
25     using {
26         nameAlt : String = thisModule.getStatement.master.name;
27         locationAlt : String = thisModule.getStatement.master.path;
28     }
29     to t : OUTConfiguration!Environment (
30         name <- s.name,
31         runningServices <- s.runningServices,
32         ...
33       )
34     ...
35     do {
36       if (thisModule.existsFile(locationAlt)) {
37         if (thisModule.existsAlternative(nameAlt)) {
38           thisModule.getAlternative(nameAlt).current <- thisModule.getFile(locationAlt);
39           t.alternatives <- INConfiguration!Alternative.allInstancesFrom('IN')->collect(
                 ↪e | thisModule.Alternative(e));
40         } else {
41           t.alternatives <- INConfiguration!Alternative.allInstancesFrom('IN')->collect(
                 ↪e | thisModule.Alternative(e)).asSequence().including(thisModule.
                 ↪createAlternative(nameAlt, thisModule.getFile(locationAlt)));
42         }
```

```
43        } else {
44          t.alternatives <- INConfiguration!Alternative.allInstancesFrom('IN')->collect(e
              ↪| thisModule.Alternative(e));
45          thisModule.createErrorModel('add_alternative', 'The␣file␣'+ locationAlt + '␣does
              ↪␣not␣exist!');
46        }
47      }
48  }
```

Listing 4.1: Fragment of the ATL transformation defining the semantics of the `addAlternative` statement

ATL does not provide a native support to compose different transformations; they are typically composed by means of Ant [Apa] scripts that are difficult to manage. To overcome such difficulties, the upgrade simulator adopts Wires* [RRGLR+09] a domain specific language that enables the high-level orchestration of model transformations. Wires* has also an execution engine able to load models and execute transformation chains according to the source Wires* model. Thus to simulate the execution of a maintainer script, by means of the `OrchestrationModelManager` component in Figure 4.11, the simulator (i) creates at run-time a Wires* model representing the composition of the ATL transformations related to the statements to be simulated, and (ii) executes the composed model transformations according to the created Wires* model (see the method `runOrchestrationModel` of the `OrchestrationModelManager` component).

The `SimulatorController` in Figure 4.11 is in charge of controlling the overall simulation process by enacting the proper upgrade scenario according to the current configuration. Listing 4.2 shows a fragment of the `start` method of `SimulatorController` that manages the simulation phases according to the provided upgrade plan and the current configuration. In particular, for each package specified in the upgrade plan (see line 3 of Listing 4.2) a number of checks are performed.

If the action to be simulated is the *installation* of the package (see line 9) then the `SimulatorController` checks the current state of the package in the configuration model. If the package is already installed then the package upgrade scenario (shown in Appendix A) is simulated (see lines 10-11). If the package is in the `Config-files` state, the install scenario shown in Figure 4.4 is simulated (see lines 12-13). If the package is in the `Half Installed` state the reinstallation of the package is simulated (see Section 4.2.3). If the package is in the `Half-Configured` state, the reinstallation of the package is simulated (see line 17-19). Finally, if the package is not installed, the upgrade scenario in Figure 4.3 is simulated (see lines 20-21). If the `install` action is not allowed in the current state of the considered package an error is raised (see lines 22-24).

If the action to be simulated is the *removal* of the considered package (see line 26), the upgrade scenario in Figure 4.6 is simulated if the considered package is in the `InstalledPackage` state (see lines 27-28). If the package is in the `Half Installed` or `Half-Configured` states the package reinstallation and then its removal are simulated (see lines 29-33). In all the other cases an error is raised (see lines 34-35).

Finally, if the action to be simulated is `purge` (see line 37), the upgrade scenario in Figure 4.8 is simulated if the considered package is in the `Installed` state (see lines 38-39). If the package is in the state `Config-files` the scenario in Figure 4.7 is simulated (see lines 40-41). If the package is in the `Half installed` state the package reinstallation and its purge are simulated (see lines 42-46). In all the other cases, an error is raised.

```
1  public void start() throws Exception {
2    ...
3    for (int i = 0; i < sequencePkg.getSizePackageSequence(); i++) {
4
5      String action = sequencePkg.getPackageAction(i);
```

```
6     SystemModelManager sysModel = new SystemModelManager();
7     SimulatorContext p;
8
9     if (action.equals("install")) {
10      if (sysModel.isInstalledPackage(sequencePkg.getPackageName(i))) {
11        // upgrade + reinstall
12      } else if (sysModel.isConfigFilesPackage(sequencePkg.getPackageName(i))) {
13        // install from configFiles state
14      } else if (sysModel.isHalfInstalledReinstRequiredPackage((sequencePkg.
            ↪getPackageName(i)))) {
15        // reinstall from half-installed state
16        // (Reinst-Required)
17      } else if (sysModel.isHalfConfiguredReinstRequiredPackage((sequencePkg.
            ↪getPackageName(i)))) {
18        // reinstall from half-configured
19        // (= Config-failed) state (Reinst-Required)
20      } else if (sysModel.isNotInstalled(sequencePkg.getPackageName(i) {
21        // simple case: install from NotInstalledState
22      } else {
23        // The requested selection-state is not allowed
24        // on the current state of the package
25      }
26    } else if (action.equals("remove")) {
27        if (sysModel.isInstalledPackage(sequencePkg.getPackageName(i))) {
28            // Remove from Installed state
29          } else if (sysModel.isHalfInstalledReinstRequiredPackage((sequencePkg.
                ↪getPackageName(i)))
30              || sysModel.isHalfConfiguredReinstRequiredPackage((sequencePkg.
                    ↪getPackageName(i)))) {
31              // reinstall from half-installed/half-configured
32              // state + remove (Reinst-Required)
33              } else {
34                // The requested selection-state is not allowed
35                // on the current state of the package
36        }
37    } else if (action.equals("purge")) {
38        if (sysModel.isInstalledPackage(sequencePkg.getPackageName(i))) {
39          // Purge from Installed state
40        } else if (sysModel.isConfigFilesPackage(sequencePkg.getPackageName(i))) {
41          // Purge from ConfigFiles state
42        } else if (sysModel.isHalfInstalledReinstRequiredPackage((sequencePkg.
              ↪getPackageName(i)))
43            || sysModel.isHalfConfiguredReinstRequiredPackage((sequencePkg.
                  ↪getPackageName(i)))) {
44          // reinstall from half-installed/
45          // half-configured state + purge
46          // (Reinst-Required)
47        } else {
48          // The requested selection-state is not allowed
49          // on the current state of the package
50        }
51    } else {
52      // The requested selection-state is not allowed
53      // on the current state of the package
54    }
55  }
56 }
```

Listing 4.2: Fragment of the `SimulatorController`

Next section provides more insights about the execution of the simulator on real examples by considering the building blocks previously described.

4.4.2 Running the Upgrade Simulator

As said in the previous section, given a source configuration model the simulator is able to simulate a given upgrade plan to notice possible problems in advance without affecting the

real system. The system configuration and the package models are automatically retrieved by the simulator by using the `ConfigurationInjector` and the `PackageInjector` components, respectively (see Figure 4.11). This section discusses in detail how the upgrade plan is given (see Section 4.4.2), and how it is simulated (see Section 4.4.2). Section 4.4.2 provides more insights about the simulation of the maintainer scripts involved in the considered upgrades.

Specification of the upgrade plan

The upgrade plan that has to be simulated is represented in terms of an XML document, like the one reported in Listing 4.3. This XML document for the upgrade plan is automatically produced by MPM (Mancoosi Package Manager) [ACTZ11], which is a modular package manager realized within the Mancoosi project . It allows the user to specify high-level user-defined optimization criteria, which are used to choose an installation solution well adapted to the user needs, and which is computed by resorting to an efficient external solver.

Essentially, the document contains a sequence of operations (`install`/`remove`/ `purge`) executed on packages each represented by means of name, version, and target architecture. For instance, the first operation in the document in Listing 4.3 is the installation of the package `swi-prolog` - version `5.6.58`, which is usable on any architecture. The removal of the same package is represented at lines 8-12 of Listing 4.3 that induces the removal of the `swi-prolog` package while maintaining its configuration files. Such files will be considered when re-installing the package as done at lines 13-17. In case of a `purge` operation also the configuration files of the considered package are removed. Lines 18-22 show the installation of the package `swi-prolog` of version `5.7.59`; this corresponds to the upgrade of the already installed version `5.6.58`. It is important to note that having the upgrade plan specified as an XML document permits the simulator to be independent from the planner that has been adopted to satisfy the upgrade request. More information about this aspect will be provided in Section 4.5 where we present the integration of the simulator with real Linux distributions.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <selectionStates  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
3    <selectionState type="install">
4         <param name="package" value="swi-prolog"/>
5         <param name="version" value="5.6.58"/>
6         <param name="architecture" value="all"/>
7    </selectionState>
8    <selectionState type="remove">
9         <param name="package" value="swi-prolog"/>
10        <param name="version" value="5.6.58"/>
11        <param name="architecture" value="all"/>
12   </selectionState>
13   <selectionState type="install">
14        <param name="package" value="swi-prolog"/>
15        <param name="version" value="5.6.58"/>
16        <param name="architecture" value="all"/>
17   </selectionState>
18   <selectionState type="install">
19        <param name="package" value="swi-prolog"/>
20        <param name="version" value="5.7.59"/>
21        <param name="architecture" value="all"/>
22   </selectionState>
23 </selectionStates>
```

Listing 4.3: Sample upgrade plan

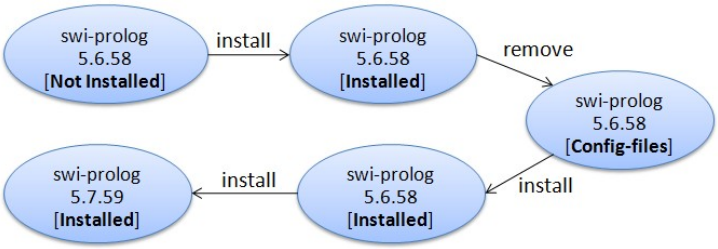


Figure 4.12: States of the `swi-prolog` package during the sample upgrade plan in Listing 4.3

Simulation of the upgrade plan

Simulating upgrade plans, like the one in Listing 4.3, consists of executing the sequence of the specified operations on the source configuration model. The execution of each operation depends on the state that the considered package has in the source configuration. For instance, the different states of the sample package `swi-prolog` during the execution of the upgrade plan in Listing 4.3 are reported in Figure 4.12. In particular, by referring to the upgrade scenarios discussed in the previous section, the first `install` operation corresponds to the *Installation performed in a Not installed state* in Figure 4.3, since the package `swi-prolog` is not installed in the considered source configuration. Then the `remove` operation corresponds to the *Removal performed in an Installed state* in Figure 4.6 and leads the package `swi-prolog` in the `Config-files` state. The re-installation of the just removed package corresponds to the *Installation performed in a Config-Files state* scenario shown in Figure 4.4. The last `install` operation, which takes into account the version `5.7.59` of the `swi-prolog` package, corresponds to the *Package upgrade* scenario discussed in Section 4.2.4 and shown in Figure 4.5. In fact, the `swi-prolog` package is in the `Installed` state and the installation of a newer version corresponds to the upgrade of the already installed one.

Figure 4.13 shows a sequence diagram which provides details about the messages which are exchanged between the simulator components to simulate the installation of a package that is in the *Not installed* state.

Simulation of the maintainer scripts

As previously said, during the simulation of upgrade plans also the maintainer scripts of the considered packages are simulated. For instance, Figure 3.8 shows a fragment of the `swi-prolog 5.7.59` package model. Besides the files which will be added in the configuration model when the package will be installed, also the statements of the maintainer scripts are represented. For instance, the package model in Figure 3.8 contains a post-installation script consisting of four commands of the EVOSS DSL, specifically: `Message`, `If`, `Message`, and `Message`. As said in the previous section, the execution of the maintainer scripts encompasses (i) the generation of a Wires* model that orchestrates the model transformations corresponding to the statements to be simulated, and (ii) the execution of the generated Wires* model.

```
1 <?xml version="1.0"?>
2 <config>
3   <mapping elementModel="PostinstAddAlternative" transformationFile="add_alternative.atl
      ↪"/>
4   <mapping elementModel="Message" transformationFile="default_copy.atl"/>
5   <mapping elementModel="If" transformationFile="conditionHOT.atl"/>
```

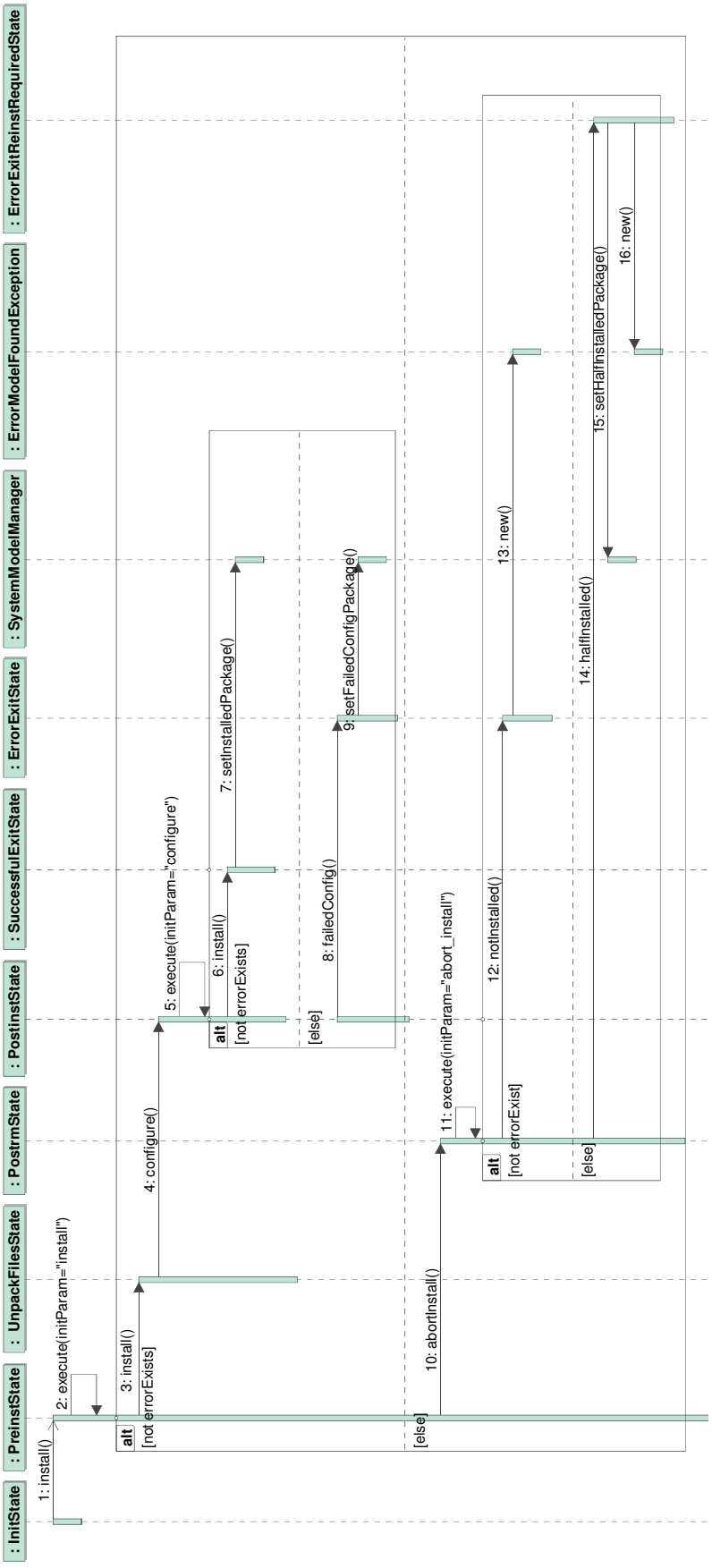


Figure 4.13: Simulation of a package installation from the *Not installed* state

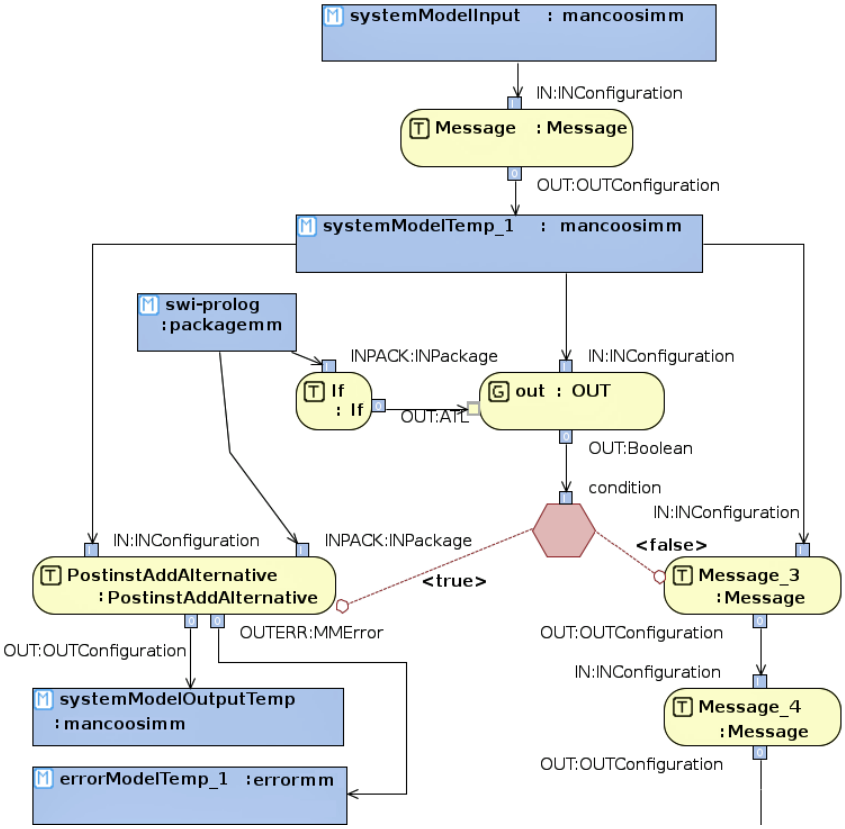


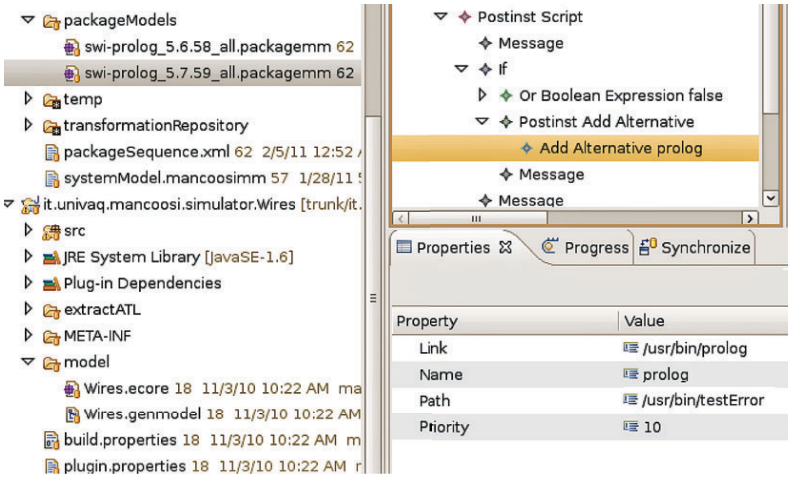
Figure 4.14: Fragment of the generated Wires* model

```
6   ...
7 </config>
```

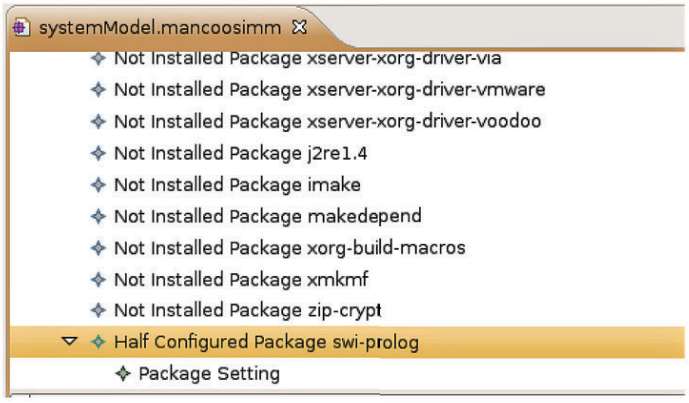
Listing 4.4: Fragment of the `mapping document`

In order to generate the Wires* model, the simulator takes as input a `mapping document` like the one in Listing 4.4; this specification relates each statement of the EVOSS DSL with a corresponding model transformation that provides its semantics. For instance, according to Listing 4.4, the semantics of the statement `PostinstAddAlternative` is given by the `add_alternative.atl` transformation. The DSL statement `Message` does not affect the source configuration model; thus in line 4 of Listing 4.4 the `default_copy.atl` transformation is specified. It generates a target configuration model, which is exactly the same of the source one. The `If` statement has the `conditionHOT.atl` transformation associated (see line 5). Such a transformation is higher-order since it generates a target ATL query whose evaluation corresponds to the expression evaluation of the source `If` statement. According to the outcome of the evaluation, the `add_alternative.atl` transformation might be executed because of the first branch of the `If`.

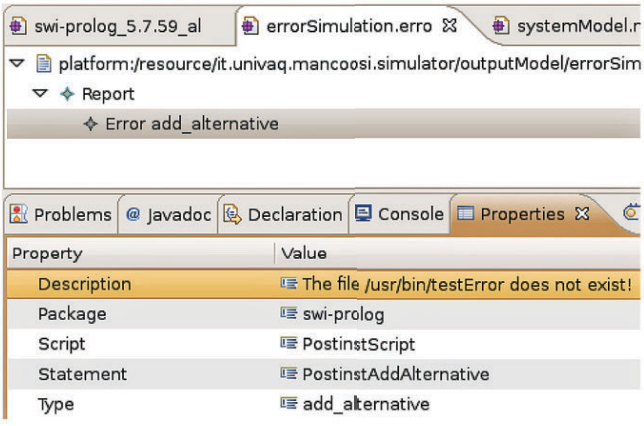
Figure 4.14 shows a fragment of the Wires* model that has been automatically generated by the simulator starting from the *postinst* script of the sample package in Figure 3.8. The model consists of boxes that represent source and target models; they are the input and the output, respectively, of transformations or queries, which are represented with rounded boxes. For instance, the simulation of the `If` statement is performed by executing the `If` transformation that generates a query named `out` in the model. Such a query is evaluated on the configuration



(a) Modified script of swi-prolog 5.7.59



(b) Generated configuration model

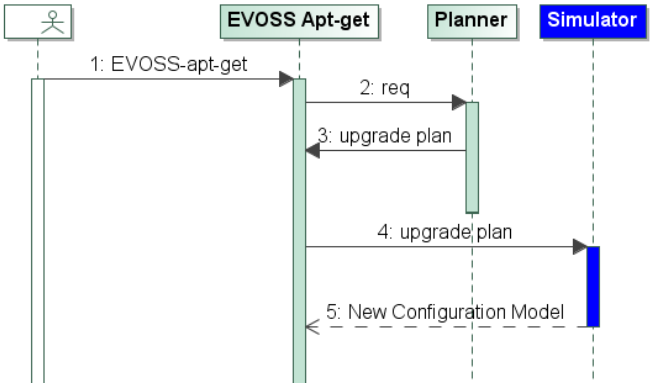


(c) Generated error model

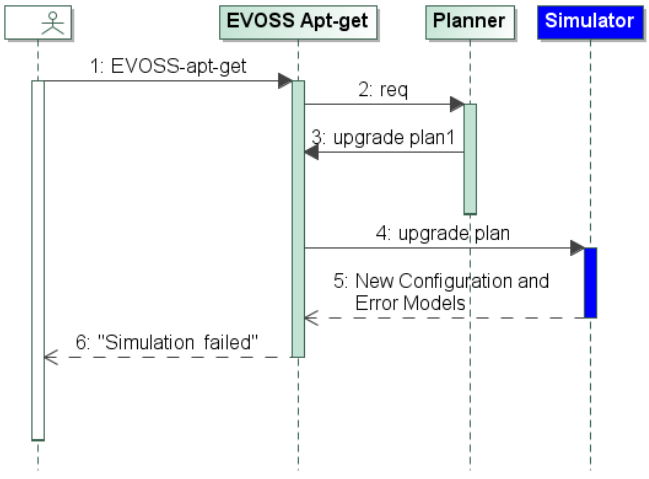
Figure 4.15: Modified input package and sample simulation outcome

model generated by the transformation related to the `Message` statement. If the query returns `true` the transformation related to `PostinstAddAlternative` is executed by generating a new configuration model and possibly an error model. If the query returns `false`, the other branch of the `If` is simulated.

A sample error model that can be generated during the upgrade simulation is shown in Figure 4.15.c. Such a model has been generated by the execution of `addAlternative.atl` transformation because of an error that we intentionally injected in the post installation script of the sample package `swi-prolog 5.7.59`. In particular, as shown in the package model in Figure 4.15.a that has been generated after the modification, the `prolog` alternative (which the script should add) refers to the executable `/usr/bin/testError`. This executable is not existing in the source configuration model, thus the simulation creates the configuration model in Figure 4.15.b and the error model in Figure 4.15.c. Because of the error in the maintainer script, after its simulation the state of the `swi-prolog` package is set as `Half-configured`.



(a) Successful simulation



(b) Failed simulation

Figure 4.16: Meta-installers enhanced by the Evoss simulator

4.5 Integrating the simulator with real distributions

In this section we describe how the Evoss simulator can be integrated with real distributions. Since the simulator is part of the Evoss approach it is obvious that other modules of Evoss concur on the integration of the simulator in real environments. In this section we focus on the simulation part while providing the information that is needed in order to have a clear idea of the applicability in practice of the simulator. However, we refer to [DCDRP+10] for a discussion of the application in practice of the overall Evoss approach.

The elementary operations performed by the simulator are model transformations that are executed to simulate the effect of script statements. Therefore, a key enabler for using in practice the simulator is writing the maintainer scripts in the Evoss DSL. In a future scenario we can envision a Linux distribution whose maintainer scripts are natively written in the Evoss DSL. However, in order to promote the integration with current distributions, Evoss makes use of the package injector component; this component gets as input a package together with its maintainer scripts and automatically produces a model of the package in which maintainer scripts are represented as Evoss DSL scripts.

The simulator might be invoked from command line and the upgrade plan is provided as an XML file. Figure 4.5 shows the enhancement of existing meta-installers. The figure contains three different scenarios. Figure 4.16(a) shows a successful simulation that will be followed by the real system upgrade. Figure 4.16(b) shows a failed simulation, i.e., a simulation that discovers some errors and that consequently produces an error model together with a new configuration. In this case the user is notified about the simulation result and he can then decide to continue with the upgrade or not.

4.6 Experiments

In this section we discuss some experiments we have done to validate the upgrade simulator. The aim of such experiments is to verify that the outcomes of the upgrade simulators are correct, i.e., they do not differ from the system configurations that are obtained after the concrete applications of the simulated upgrades. Section 4.6.1 discusses how we have selected the subjects that have been considered in the validation. Section 4.6.2 describes the setup process we followed to perform the simulation. Section 4.6.3 reports and discusses the results of the experimentation. Finally, Section 4.6.4 discusses the threats of validity of our experiments.

4.6.1 Case study subjects

The case study subjects have been selected by covering the upgrade scenarios described in Section 4.2, i.e., each upgrade scenario is covered by at least one case study subject. Moreover, we considered upgrades involving upgrade plans of different dimension. Table 4.1 shows some information on the case study subjects. In particular, Table 4.1 shows package name, upgrade operation, and package state.

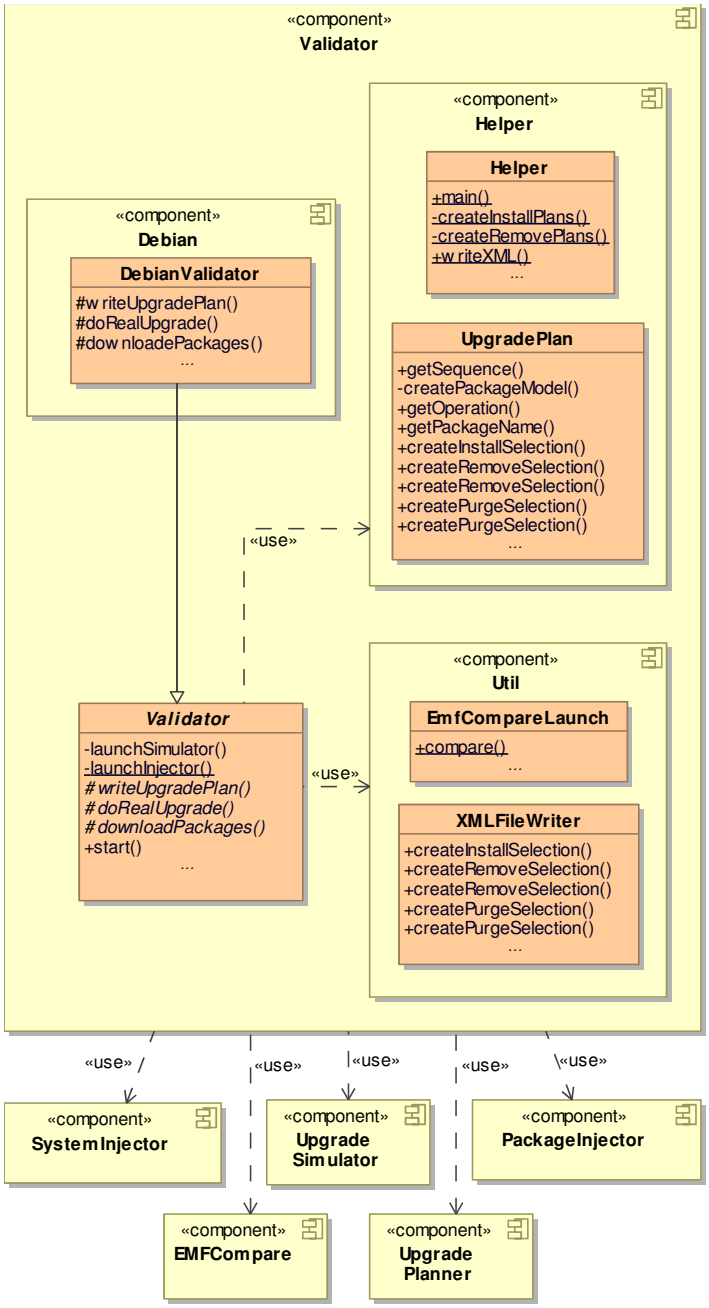


Figure 4.17: Validator architecture

	Package	Upgrade operation	Package state
1	amule	install	Not installed
2	compiz	install	Not installed
3	filezilla	install	Not installed
4	gnome-utils	install	Not installed
5	mysql-admin	install	Not installed
6	remmina	install	Not installed
7	stardict	install	Not installed
8	vino	install	Not installed
9	alsa-oss	install	Config-Files
10	freeglut3	install	Config-Files
11	libxdot4	install	Half-installed (reinst-required)
12	libspeex1	install	Half-configured (reinst-required)
13	libgadu3	install	Installed
14	libdiscid0	remove	Installed
15	libcln6	remove	Installed
16	iptables	remove	Half-installed (reinst-required)
17	konq-plugins	remove	Half-configured (reinst-required)
18	libgeoclue0	purge	Installed
19	libgpgme++2	purge	Installed
20	libsrtp0	purge	Installed
21	mtr	purge	Half-Configured (reinst-required)
22	bug-buddy	purge	Config-Files
23	reportbug	purge	Half-configured (reinst-required)

Table 4.1: Case study subjects

4.6.2 Experimental setup

In order to perform the experiment we implemented the `Validator` component shown in Figure 4.17 that helped us to automatize the overall validation process. Such a component gets as input an upgrade request (like the ones shown in Table 4.1) composed of a package name and an upgrade operation and, according to Figure 4.18, performs the following steps:

1. Invokes the `UpgradePlanner` component to obtain an upgrade plan which satisfies the upgrade request. More specifically it makes use of `apt-get` package manager[2] executed with the option `--simulate` that calculates the upgrade plan without affecting the real system.

2. By means of the `PackageInjector` component generates the models of the packages involved in the upgrade plan if they are not already available.

3. By means of the `SystemInjector` component `Validator` generates the model of the running system (named CM_1).

4. The upgrade is simulated by using the `UpgradeSimulator` component and the new configuration model CM_2 is generated.

[2] `http://linux.die.net/man/8/apt-get`

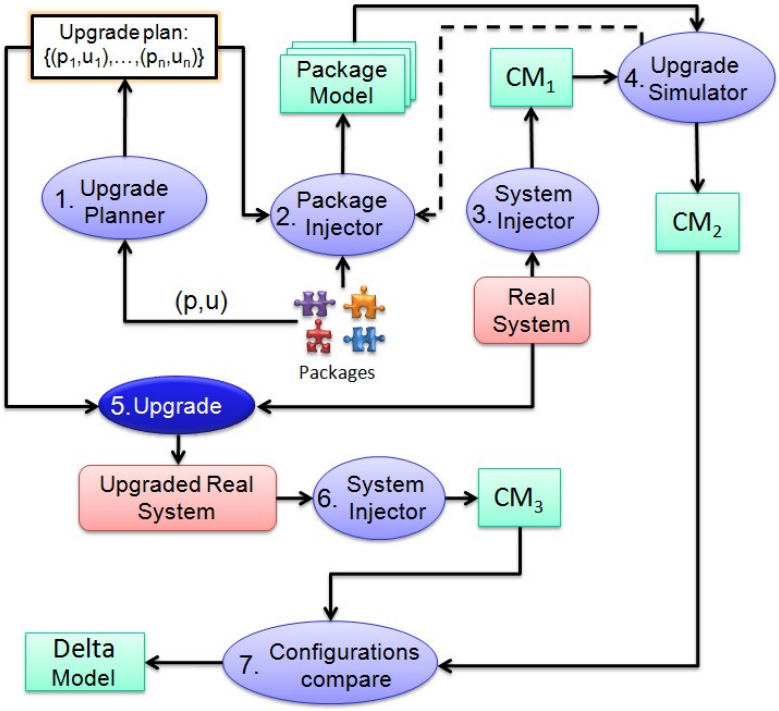


Figure 4.18: Experimental setup

5. The real upgrade of the system is performed, thus producing a new *Upgraded Real System*.

6. The `SystemInjector` component is used again to obtain from the upgraded system a new configuration model CM_3.

7. Finally, the *EMFCompare*[3] component is used to check if CM_2 and CM_3 represent exactly the same configuration. To this purpose the `Validator` component generates a delta model containing the difference between the two models. If the delta model is empty then the two models are equal. It is worth mentioning that the comparison step has been performed by using the generic matching algorithm of *EMFCompare* without the need to customize it.

The experiment has been performed on an Intel Core2 Duo CPU P8600 - 2.40 GHz, 4 GB of RAM, with installed a Debian Squeeze distribution.

4.6.3 Results

Table 4.2 reports the results of the validation. As can be seen, the validation reports a *valid* result for any case study subject. This means that the configuration model produced by the simulator (when the simulation has been performed) is equal, according to the differencing operation performed by EMFCompare, to the configuration model obtained by injecting the upgraded system. For each upgrade we show also the number of upgrade plan operations required (as produced by apt-get). As can be seen some upgrades involve many operations;

[3] `http://wiki.eclipse.org/index.php/EMF_Compare`

Case study subject	Experiment result	# of upgrade plan operations	# of involved maint. scripts	Execution time (simulator)
1	Valid	4	3	11.84 s
2	Valid	13	6	24.13 s
3	Valid	6	5	15.1 s
4	Valid	14	11	32.45 s
5	Valid	8	7	19.8 s
6	Valid	5	4	13.5 s
7	Valid	3	1	12.13 s
8	Valid	2	2	8.18 s
9	Valid	1	1	4.05 s
10	Valid	1	1	4.13 s
11	Valid	1	1	4.22 s
12	Valid	1	1	3.53 s
13	Valid	1	2	8.12 s
14	Valid	3	4	10.34 s
15	Valid	4	3	11.33 s
16	Valid	1	2	6.54 s
17	Valid	1	2	6.89 s
18	Valid	9	2	28.16 s
19	Valid	8	14	72.35 s
20	Valid	7	14	68.61 s
21	Valid	1	3	11.43 s
22	Valid	1	1	3.56 s
23	Valid	1	3	10.21 s

Table 4.2: Experiment results

for example, the upgrade plan for installing `gnome-utils` - case study subject n.4 - (when the package is in a `Not installed` state) involves 14 operations. Listing 4.5 reports the upgrade plan for the installation of the package `gnome-utils` when it is in the `Not installed` state.

```
<?xml version="1.0" encoding="UTF-8"?>
<selectionStates
  xmlns="http://www.w3.org/2001/XMLSchema-instance">
  <selectionState type="install">
    <param name="package" value="gnome-utils-common"/>
    <param name="version" value="2.30.0-2"/>
    <param name="architecture" value="all"/>
  </selectionState>
  <selectionState type="install">
    <param name="package" value="baobab"/>
    <param name="version" value="2.30.0-2"/>
    <param name="architecture" value="i386"/>
  </selectionState>
  <selectionState type="install">
    <param name="package" value="libgdict-1.0-6"/>
    <param name="version" value="2.30.0-2"/>
    <param name="architecture" value="i386"/>
  </selectionState>
  <selectionState type="install">
    <param name="package" value="libpanel-applet2-0"/>
    <param name="version" value="2.30.0-2"/>
    <param name="architecture" value="i386"/>
  </selectionState>
  <selectionState type="install">
    <param name="package" value="gnome-dictionary"/>
```

```
26      <param name="version" value="2.30.0-2"/>
27      <param name="architecture" value="i386"/>
28    </selectionState>
29    <selectionState type="install">
30      <param name="package" value="libtdb1"/>
31      <param name="version" value="1.2.1-2+b1"/>
32      <param name="architecture" value="i386"/>
33    </selectionState>
34    <selectionState type="install">
35      <param name="package" value="libvorbisfile3"/>
36      <param name="version" value="1.3.1-1"/>
37      <param name="architecture" value="i386"/>
38    </selectionState>
39    <selectionState type="install">
40      <param name="package" value="libcanberra0"/>
41      <param name="version" value="0.24-1"/>
42      <param name="architecture" value="i386"/>
43    </selectionState>
44    <selectionState type="install">
45      <param name="package" value="libcanberra-gtk0"/>
46      <param name="version" value="0.24-1"/>
47      <param name="architecture" value="i386"/>
48    </selectionState>
49    <selectionState type="install">
50      <param name="package" value="gnome-screenshot"/>
51      <param name="version" value="2.30.0-2"/>
52      <param name="architecture" value="i386"/>
53    </selectionState>
54    <selectionState type="install">
55      <param name="package" value="gnome-search-tool"/>
56      <param name="version" value="2.30.0-2"/>
57      <param name="architecture" value="i386"/>
58    </selectionState>
59    <selectionState type="install">
60      <param name="package" value="gnome-system-log"/>
61      <param name="version" value="2.30.0-2"/>
62      <param name="architecture" value="i386"/>
63    </selectionState>
64    <selectionState type="install">
65      <param name="package" value="gnome-utils"/>
66      <param name="version" value="2.30.0-2"/>
67      <param name="architecture" value="all"/>
68    </selectionState>
69    <selectionState type="install">
70      <param name="package" value="libcanberra-gtk-module"/>
71      <param name="version" value="0.24-1"/>
72      <param name="architecture" value="i386"/>
73    </selectionState>
74  </selectionStates>
```

Listing 4.5: Upgrade plan for the oparation `Install` of the package `gnome-utils` when it is in the `Not installed` state

It is important to note that this validation involves also other components of the EVOSS approach, namely the package injector and the system injector. However, the performance evaluation, as shown in table, only refers to the EVOSS simulator. We report also the involved maintainer scripts[4] since this can help understanding the performance of each upgrade. As can be seen the most expensive upgrade, i.e., the upgrade of `libgpgme++2` - case study subject n.18, requires 1 minute and 12 seconds. This upgrade is quite complex since it involves 8 operations and requires the execution of 14 maintainer scripts. Typically upgrades are more simple. We considered these cases to show that it is reasonable to use the upgrade simulator in practice.

[4]We recall that a package is not obliged to define a maintainer script if it is not necessary. For instance, as shown in Table 4.2, the installation of `amule` - case study subject n.1 - while involving 4 operations, requires the execution of only 3 maintainer scripts.

4.6.4 Threats to Validity

The results of our experiments are subjects to the following threats of validity:

- The packages we have investigated comes from a real distribution; while this is a large case study, we cannot claim that the results of our experimental evaluation are generalizable. The evaluation should be considered as investigating the potential of the technique rather than providing a statement of effectiveness. However, the performed validation provide us an acceptable confidence of the accuracy and correctness of the simulator.
- The entire Evoss approach and then also its simulator component are engineered to be easy generalizable to other distributions. We are instantiating the approach to the Caixa Magíca RPM distribution[5].
- The tools we have used or implemented could be defective. To counter this threat, we have run several manual assessments and counter-checks.
- We also evaluated the performance of the simulator and we show in Table 4.2 a summary of the results. This aspect is important for its use in practice.

[5] `http://caixamagica.pt`

Chapter 5

Fault Detector

Given a system configuration model, the *fault detector* [DP11] checks the model in order to identify possible faults. To analyze the system configuration model, specific query expressions are evaluated, each representing a class of faults. The language used for expressing the queries is OCL. It is a declarative language that provides constraints and object query expressions on models and meta-models. An example of OCL query is shown in Listing 5.1: given a configuration model (`IN` conforming to the metamodel `MM`), all the instances of the `MimeTypeHandler` metaclass are retrieved and those that do not have a handler specified are considered. When the result of this query is greater than 0, a fault is detected. In fact, such configurations are considered to be incoherent due to the existence of mime types without corresponding handlers installed (e.g., the owning package has been deleted without un-registering the handler).

Listing 5.1: Fragment of the OCL query to detect missing mime type handlers

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

Since the system configuration model is an abstraction of the real system, there are some elements of the real system which are not directly represented in the model. Thus, there are some faults which require both the system model and the running system to be identified. For instance, according to a system configuration model, the corresponding system should be able to open PDF (Portable Document Format) files. Even though this information is available in the model, it is necessary to check if a PDF editor is really installed in the running system.

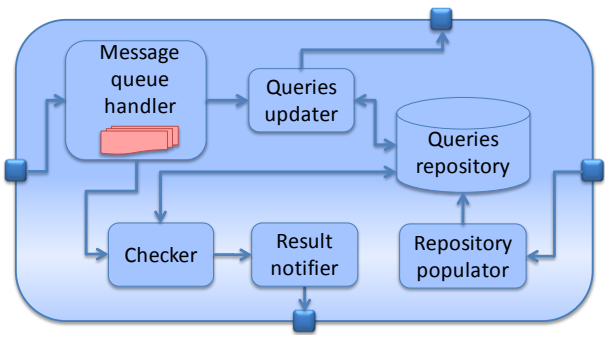


Figure 5.1: Server overview

The Evoss fault detector permits to specify such queries by using Java and embodying them in JAR files.

The fault detector is extensible in the sense that when new classes of faults are identified, a corresponding query can be defined and added to it. In particular, we realized the fault detector by using the client-server architectural style. In this way different user communities can contribute in the definition of queries and store them in a centralized manner. Then, each user can increase his fault detection capabilities by retrieving updated queries from the server.

Figure 5.1 shows an overview of the server-side of the fault detector. It consists of six main components described in the following. The `Message queue handler` component receives requests from clients and forward them to the `Queries updater`, or to the `Checker` depending on the client needs. In particular, the former sends back to the client the updated set of queries, which can be evaluated locally by the client. The latter evaluates on the system configuration model of the client the queries available in the `OCL queries repository`, and produces a result that is notified by the `Result notifier` component. Each query can have associated a problem solution that is proposed to the user if the execution of the query returns an error. The community may populate the `OCL queries repository` with new queries by means of the `Repository populator` component.

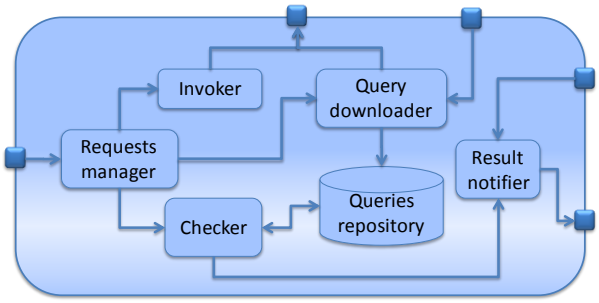


Figure 5.2: Client overview

An overview of the client of the fault detector is shown in Figure 5.2. It permits to realize the usage scenarios presented in the remaining of the section. In particular, a user can decide to execute the `Checker` of the server or the one available in the client. They are slightly different: the client's checker can evaluate queries that require also system information. This is not possible on the server side since the server gets as input only the model of a system configuration. Therefore, the fact of executing the checker on the client machine enables the execution of more sophisticated queries, such as queries that require to check the presence of a file. The component that manages the different kinds of requests is the `Requests manager` component. The `Result notifier` component retrieves the results either from the server side or from the client side (`Checker` component) and notifies the user.

Fault detector usage scenarios

In this section we describe some representative scenarios of the fault detector. More precisely, we present four scenarios in which are involved the client and the server. For a detailed list of the functionalities of the failure detector, please refer to the user guide available at `http://evoss.di.univaq.it/`.

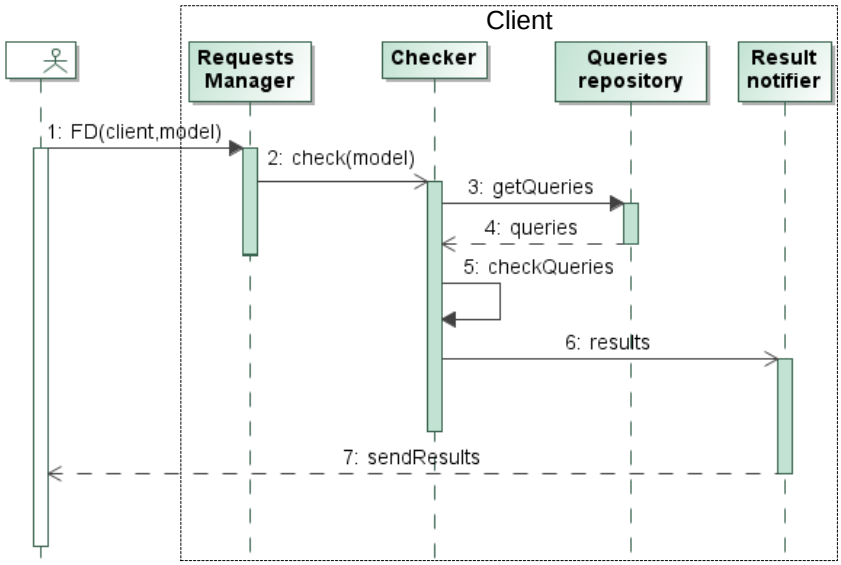


Figure 5.3: First scenario: the user invokes the checker that is on the client side

- *Scenario 1*: As shown in Figure 5.3, the user invokes the checker that is available on the client side (the `Checker` component). All the queries that are available on the `Queries repository` of the client are evaluated. An alternative of this scenario is to update the queries repository of the client before performing the check.

- *Scenario 2*: As shown in Figure 5.4, the community might add, modify, or delete queries of the `Queries repository` of the server. The interaction can be realized by using a Web portal.

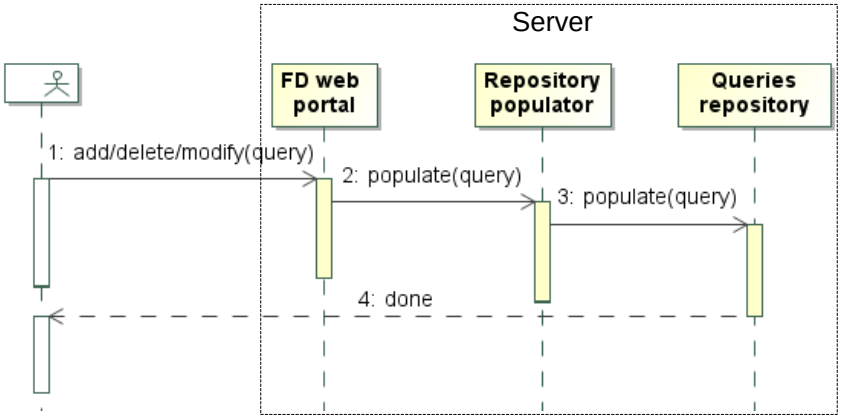


Figure 5.4: Second scenario: populating the `Queries repository` of the server

- *Scenario 3*: As shown in Figure 5.5, by means of the `Invoker` component, the user invokes the checker that is available on the server by passing the model. User requests are stored in a message queue. The handling of a request consists of running on the model each OCL query included in the queries repository (we recall that the server cannot execute JAR queries since the running client machine is not accessible from the server).

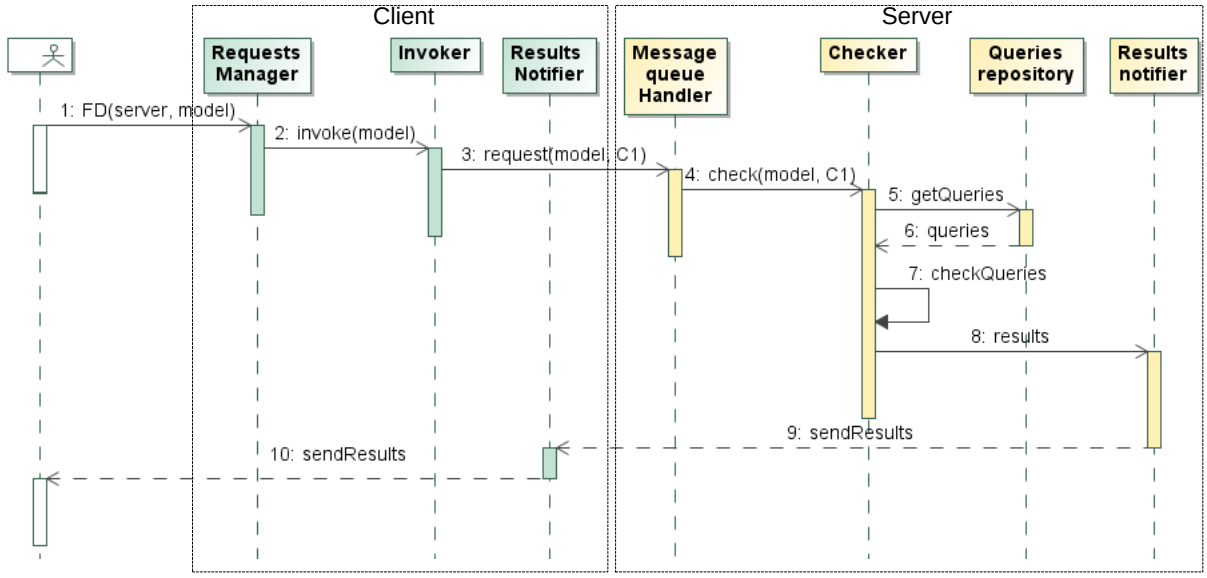


Figure 5.5: Third scenario: the user invokes the checker that is on the server side

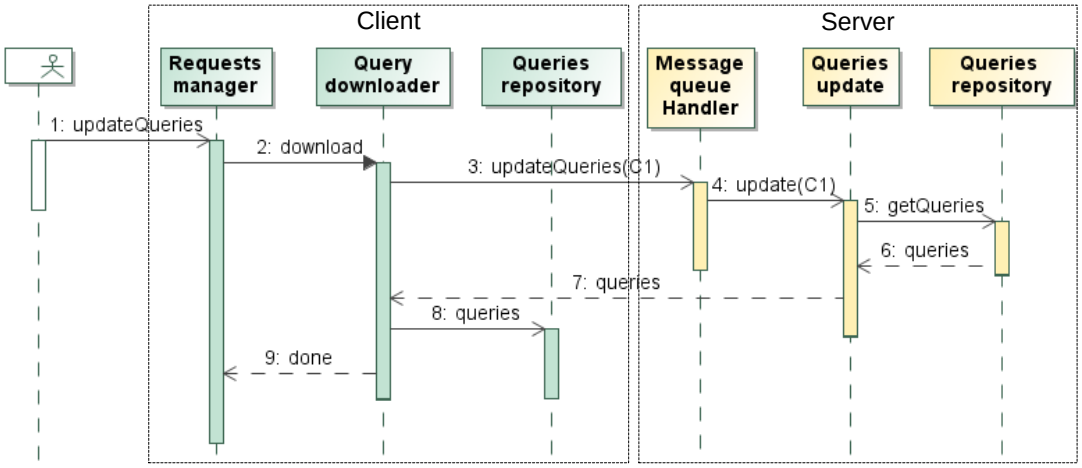


Figure 5.6: Fourth scenario: OCL queries update

- *Scenario 4*: As shown in Figure 5.6, the user requests to just update the set of queries. This is realized by means of the `Query downloader` component that adds the request to the message queue of the server. Then, the server retrieves the queries from the queries repository and sends the queries back to the client.

5.1 Implementing the fault detector

In this section we present an implementation of the fault detector. The fault detector has been conceived by using different model-driven techniques and tools based on the Eclipse Modeling Framework (EMF) [BSM+03]. The implementation of the fault detector together with the implementation of the whole EVOSS approach is freely available at `http://evoss.di.univaq.it`.

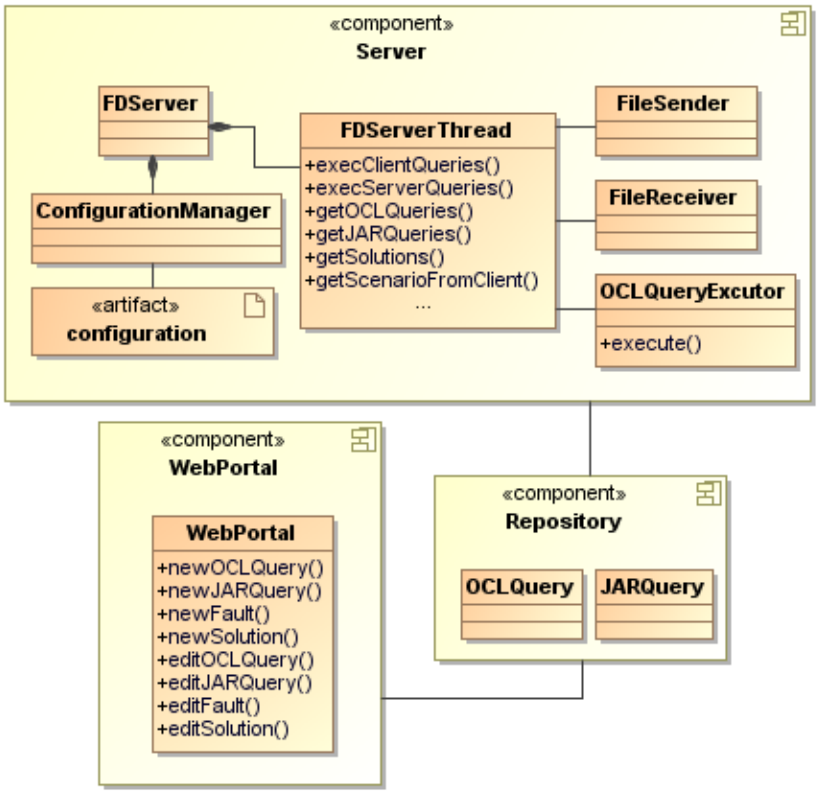


Figure 5.7: Architecture of the server-side of the fault detector

5.1.1 Architecture

As said in the previous section, the fault detector is based on a client/server infrastructure in order to realize the envisioned scenario consisting of a community, which can contribute in the definition of queries for detecting unforeseen faults. In the following the building blocks of both the client-side and the fault detector server-side are described.

Fault detector server

The architecture of the fault detector server-side is shown in Figure 5.7. It consists of three main components, named `Repository`, `WebPortal`, and `FDServer`.

Repository: It is the component that manages the persistence of the elements that are involved in the fault detection. Figure 5.8 shows the tables of the database in the repository. In particular, the repository maintains a catalog of `Faults` that are contributed by a community of `Users`. According to the previous section, faults can be discovered by querying only the system configuration model, or by considering also the running system. In this respect, the repository maintains both `OCL` and `JAR` queries. Finally, each fault can be related with one or more `Solutions`, which can be suggested to users in order to solve the discovered problems.

WebPortal: It is the component that permits to manage in a distributed manner the data stored in the repository previously presented. Whenever a new fault is discovered by the community,

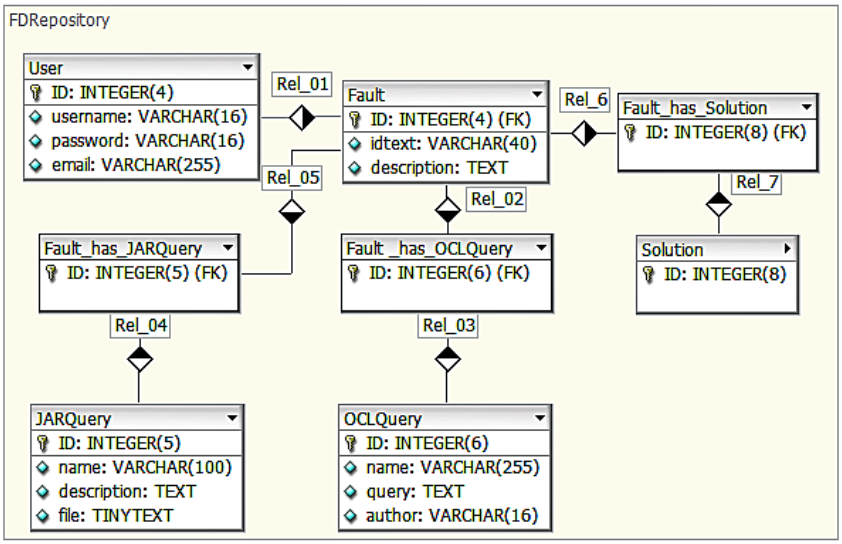


Figure 5.8: Database tables

it can be added in the repository together with (i) the queries that are able to detect it, and (ii) the description of the actions, which can be undertaken to solve the specified fault.

FDServer: It is the class that is encharged of interacting with user clients that want to analyze their systems. According to the scenarios presented in the previous section, clients can send to the server the configuration models to be analyzed, or alternatively can ask the server for the updated queries which will be executed locally, client-side. In order to manage multiple client requests, the server is implemented by using Java threads by means of the class `FDServerThread` in Figure 5.7. Some of the methods provided by such a class are the following:

- `execClientQueries`: this method permits to use the server as an executor of OCL queries the client can send to the server together with the model on which such queries have to be executed;
- `execServerQueries`: it permits to execute on the model sent by the client all the queries that are stored in the server repository;
- `getOCLQueries`: it is used by the client to update its OCL queries, which can be executed locally later;
- `getJARQueries`: it is used by the client to update its JAR queries, which can be executed locally later;
- `getSolutions`: it permits to retrieve from the repository, the solutions that can be adopted to solve faults;
- `getScenarioFromClient`: this method is used by the server to select the scenario to be executed (e.g., one of those presented in the previous section) according to the request of the client.

The execution of the previous methods involves the management of different files (i.e., the configuration model, and the files containing OCL and JAR queries), which are exchanged between the client and the server. The classes `FileSender` and `FileReceiver` implement the

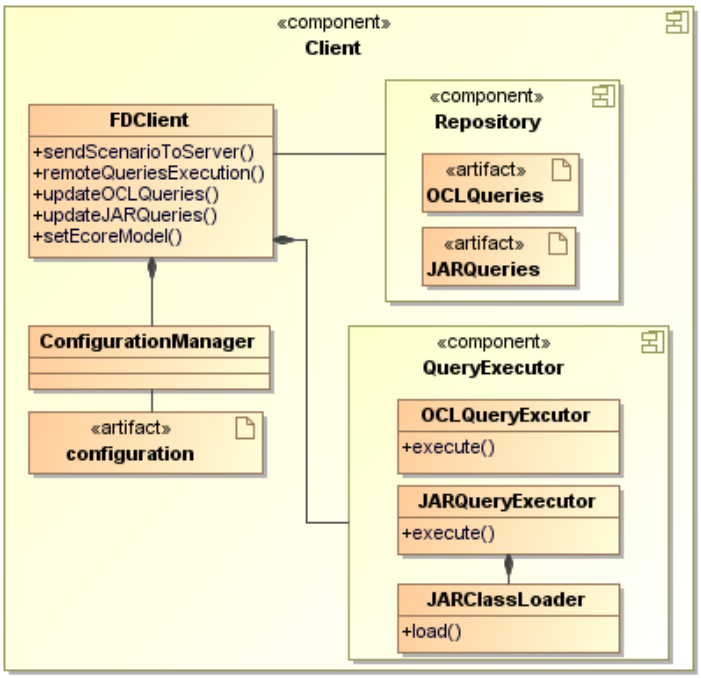


Figure 5.9: Architecture of the client-side of the fault detector

protocol underpinning such an operation. Finally, the execution of the OCL queries is performed by means of the `execute` method of the `OCLQueryExecutor` class.

Fault detector client

The architecture of the fault detector client-side is shown in Figure 5.9. The main elements are the classes `FDClient`, `Repository`, and the `QueryExecutor`.

FDClient: It is the main class, which is devoted to the interaction with the server-side. Depending on the scenario to be executed, the client can ask the server to analyze a configuration model by using the queries in the server. Alternatively, the client can update the local repository of the queries and analyze the system on its own. In addition of the OCL queries, the client can analyze the system also by means of `JARQueries`. In this case, the consistency between the configuration model and the real system can be checked. The execution of the OCL and of the JAR queries is performed by means of corresponding executors (i.e., the `OCLQueryExecutor` and the `JARQueryExecutor`, respectively).

Repository: It maintains in the file system of the client, the OCL and the JAR queries that can be used to analyze locally a system configuration model. As previously said, before the analysis, the client can update such queries and merge them with those available in the server.

QueryExecutor: The local execution of the queries is based on two classes, named `OCLQueryExecutor`, and `JARQueryExecutor`. The former makes use of MDT OCL[1] which is an implementation of the OMG OCL for EMF-based models. The JARQueryExecutor is able to load a JAR file containing a Java class implementing an interface consisting of the `run` method.

[1] http://www.eclipse.org/modeling/mdt/?project=ocl

Next section provides more insights about the execution of the fault detector on real examples by considering the building blocks previously described.

5.2 Experiments

In this section we discuss some experiments we have done to validate the fault detector. The aim of such experiments is to show how the fault detector works in practice and how it performs. Section 5.2.1 presents the case study subjects that have been considered for the experiment. Section 5.2.2 describes the setup we performed for realizing the experiment. Section 5.2.3 reports and discusses the results of the experimentation. Finally, Section 5.2.4 discusses the threats of validity of our experiments.

5.2.1 Case study subjects

In this section we describe the case study subjects that we selected for the experiment. We selected those that are more significant and easy to understand.

Missing packages involved in implicit package dependencies

In order to explain this query, let us consider the Web server `Apache2`, and `PHP5`, which is a scripting language integrated with Apache. In particular, let us assume that we want to remove the package `libapache-mod-php5` from the filesystem. Then the `PHP5` module in the Apache configuration has to be disabled before its removal. This is necessary, otherwise inconsistent configurations can be reached like the one shown in Figure 5.10. The figure reports the sample `Configuration2`, which has been reached by 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, when only `apache2` is installed. If Apache is now run as a service it may refer to the missing package, `PHP5`, which of course could lead to many errors including security breaches and services crashing.

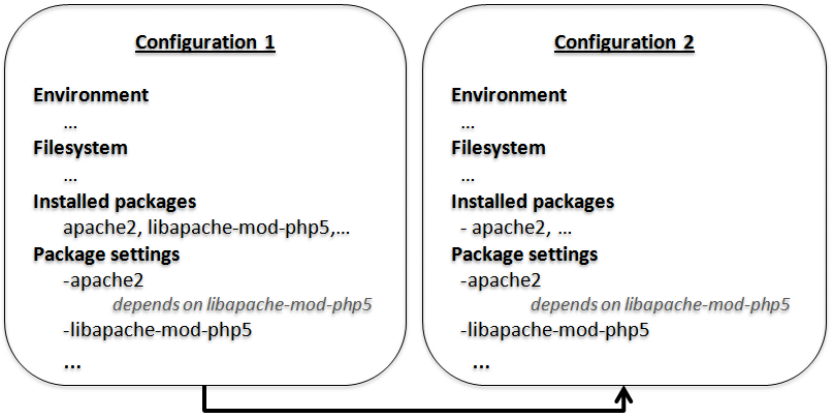


Figure 5.10: Incorrect package removal

In order to do not reach invalid configurations like the one in Figure 5.10, the package `libapache-`

mod-php5 contains the *postinst* and *prerm* scripts reported on the left hand-side and right hand-side in the following snippet, respectively.

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

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

In particular, the PHP5 module installed during the unpacking phase gets enabled invoking the a2enmod command (see line 4 on the left-hand side above); the Apache service is then reloaded (line 5) to make the change effective. Upon PHP5 removal the reverse will happen, as implemented by the *prerm* script snippet above.

It is important to note that currently, available package managers are not able to predict inconsistencies that can occur if maintainer scripts are not complete. For instance, the invalid configuration reported in Figure 5.10 can be obtained if the *prerm* above does not contain the statement 'a2dismod php5'. Package managers are not able to detect when such a statement is missing or an invalid configuration has been reached. The only types of errors they can detect are run-time failures of scripts. They cannot normally resolve the problem and perform a limited set of actions at this point. They may try and restart the configuration script, report the error to the user or, more frequently than not, they will abort the script and possibly provide an error code and/or warning message.

In general we are speaking about *implicit* dependencies among packages, i.e., dependencies that are not explicitly defined in the package meta-data. It is worth noticing that existing package managers are not able to predict inconsistencies like the one in Figure 5.10 since they take into account only information about package dependencies and conflicts expressed in the package meta-data.

Listing 5.2 reports the OCL query for checking the existence of missing packages involved in implicit package dependencies.

Listing 5.2: OCL query to detect missing packages involved in implicit package dependencies

```
1 mancoosimm::PackageSetting.allInstances()->exists( e | e.dependences->exists(d | d.
    ↪ownerPkg.oclIsTypeOf('mancoosimm::NotInstalledPackage')))
```

The solution associated with this query suggests to install the package so to fix the implicit dependencies, or to remove the package that gives place to the problem.

Existence of packages that are in the Half-installed state

Each packages has associated a state, which varies when the package is involved in an upgrade. The possible states of a package are[2]:

- Installed: the package is installed.
- Not installed: the package is not installed and the configuration files are not present in the system.

[2] According to the Debian policy http://www.debian.org/doc/debian-policy and the manpage of dpkg http://man.cx/dpkg#sec5

- `Half-installed`: when installing or removing the package some error can be experienced and then the package is left in a half-installed state. This state can also have a `Reinst required` flag meaning that the package needs to be reinstalled even if we want to remove it.

- `Config-files`: when removing a package that was previously installed, if everything goes well, the package will be left in the `Config-files` state since the configuration files of the package have not been deleted. The `purge` operation must be performed to delete also the configuration files and then to reach the `Not installed` state.

- `Half-configured`: indicates that the package has been unpacked, but its configuration scripts have failed. This state can also have a `Reinst required` flag, meaning that the package needs to be reinstalled even if we want to remove it.

- `Unpacked`: the package is unpacked, but not configured.

According to the different upgrade scenarios [DPD11], an unsuccessful upgrade might left a package in a `Half-installed` or `Half-configured` state. Such states represent possible faults which can be identified by means of the queries in Listing 5.3, and Listing 5.4.

Listing 5.3: OCL query to detect the existence of packages in the `Half-installed` state

```
1 mancoosimm::HalfInstalledPackage.allInstances()->size() > 0
```

The solution associated to this query suggests to remove or reinstall the package.

Existence of packages that are in the `Half-configured` state

The identification of faults due to the existence of packages in the `Half-configured` state can be performed by means of the query in Listing 5.4, which is similar to the one in Listing 5.3.

Listing 5.4: OCL query to detect the existence of packages in the `Half-configured` state

```
1 mancoosimm::HalfConfiguredPackage.allInstances()->size() > 0
```

The solution associated to this query suggests to remove or reinstall the package.

Existence of packages that are in the `Half-installed` state with a `Reinst required` flag

The aim of this query is to check if there exist packages in the `Half-installed` state with a `Reinst required` flag, meaning that the package needs to be reinstalled even if we want to remove it. The query is shown in Listing 5.5.

Listing 5.5: OCL query to detect the existence of packages in the `Half-installed` state with a `Reinst required` flag

```
1 mancoosimm::HalfInstalledReinstRequiredPackage.allInstances()->size() > 0
```

The solution associated to this query suggests to reinstall the package even if the user wants to remove it because of the `Reinst required` flag.

Existence of packages that are in the `Half-configured` state with a `Reinst required` flag

The aim of this query is to check if there exist packages in the `Half-configured` state with a `Reinst required` flag meaning that the package needs to be reinstalled even if we want to remove it. The query is shown in Listing 5.6.

Listing 5.6: OCL query to detect the existence of packages in the `Half-configured` state with a `Reinst required` flag

```
1 mancoosimm::HalfConfiguredReinstRequiredPackage.allInstances()->size() > 0
```

The solution associated to this query suggests to reinstall the package even if the user wants to remove it because of the `Reinst required` flag.

Missing executable in a MIME type handler specification (OCL)

MIME (Multipurpose Internet Mail Extensions) is a mechanism for encoding files and datastreams and for providing meta-information about them. Examples of these meta-information are their type (e.g., audio or video) and format (e.g., JPG, HTML, MP4). The registration of MIME type handlers allows programs, like Web browsers, to invoke the proper handlers to view, edit or display MIME types, which are not supported directly by the considered program. The aim of the query shown in Listing 5.7, is to detect if there exists a mime type handler registration, which misses the link to the executable.

Listing 5.7: OCL query to detect if executables are missing a MIME type handler specification

```
1 mancoosimm::MimeTypeHandler.allInstances()->exists(h | h.handler.oclIsUndefined())
```

The solution associated to this query suggests to link a correct executable, if this executable exists, or to remove the mime type handler registration.

Missing executable in a menu entry

The aim of this query is to check if a menu entry is missing of an executable. The OCL query is shown in Listing 5.8.

Listing 5.8: OCL query to detect `MenuEntry` elements with missing executable

```
1 mancoosimm::MenuEntry.allInstances()->exists(e | e.executable.oclIsUndefined())
```

The solution associated to this query suggests to add a correct reference to an executable or to remove the entry of the menu.

Missing executable in a Service

Linux services can be started, stopped and reloaded with the use of scripts typically stocked in `/etc/init.d/`. The aim of this query is to check if the executable of some service is missing. The OCL query is shown in Listing 5.9.

Listing 5.9: OCL query to detect service elements with missing executable

```
1 mancoosimm::Service.allInstances()->exists(e | e.executable.oclIsUndefined())
```

The solution associated to this query suggests to restore the executable or to remove the service.

Missing link in an Alternative

Alternatives is a location of symbolic links and helper-system where associations can be named and maintained in a consistent manner. It allows distribution and package owners to group and categorise packages that provide similar functionalities. For example, instead of having to refer to every email client that exists in the repositories, it is possible for package maintainers to refer to the group association "email" and, if it is required by the package, the operating system and user can select which email client they would like to use. Similarly, if a new package provides the same features as expected in the "email" category, it can use the alternatives system to suggest that it is capable of performing the actions required. Alternatives are represented with a name, which is the name of the considered alternative (e.g., `java`), and a location, which refers to the real executable in the file system (e.g., `/usr/j2se/bin/java`).

The aim of the query shown in Listing 5.10, is to check if there exist alternatives, which have the location missing.

Listing 5.10: OCL query to detect Alternative elements with missing *current* executable

```
1 mancoosimm::Alternative.allInstances()->exists(e | e.current.oclIsUndefined())
```

The solution associated to this query suggests to fix the location, if possible, to reinstall the executable required by the alternative, if needed, or to remove the alternative entry.

Existence of configuration files which can be written and/or executed by a non-root user

Configuration files must have `root` as owner. Then this query checks if there exist configuration files with owner different from `root`. The query is shown in Listing 5.11.

Listing 5.11: OCL query to detect Alternative elements with missing *current* executable

```
1 mancoosimm::PackageSetting.allInstances()->exists(e | e.files->exists(f | (f.owner.name
    ↪<> 'root')) )
```

The solution associated to this query suggests to change the owner of the configuration file.

Missing configuration files

This query analyses the running system to check if, according to the configuration model, configuration files are missing. Listing 5.12 shows the java code realizing the query.

Listing 5.12: JARQuery to detect missing configuration files

```
1  ...
2  public class MissingConfigurationFiles {
3
4    static  Configuration configuration = null;
5
6    public static void checkMissingConfigurationFiles() {
7
8      // Checking for missing files
9      System.out.println("Cheking␣for␣missing␣Configuration␣files....");
10
11     Iterator<InstalledPackage> ipi = configuration.getInstalledPackages().iterator();
12
13     while (ipi.hasNext()){
14
15       InstalledPackage ip = ipi.next();
```

```
      PackageSetting ps = ip.getPackageSettings();

      if (ps != null){
        Iterator<File> fi = ps.getFiles().iterator();
        while (fi.hasNext()){
          java.io.File aux = new java.io.File( ((File)fi.next()).getLocation() );
          if ( ! aux.exists() ) {
            System.out.println("WARNING: The configuration file " + aux.getPath()  + " is missing and it is required by the package "  + ip.getName());
          }
        }
      }
    }
    System.out.println("Cheking for missing Configuration files....DONE");
  }


  public static void main(String[] args) throws IOException{
    MancoosiPackageImpl.init();

    URI fileURI = URI.createFileURI("model/systemModel.ecore");
    Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);
    resource.load(null);
    UbuntuConfigurationManager.getInstance().setConfiguration( (Configuration) resource.getContents().get(0) );
    configuration = UbuntuConfigurationManager.getInstance().getConfiguration();

    checkMissingConfigurationFiles();
  }
}
```

The solution associated to this query suggests to restore the configuration file.

Missing executable in a MIME type handler specification (JAR)

This query checks the running system to check if executables associated to a MIME type handler specification are missing. This query is similar to the query *Missing executable in a MIME type handler specification (OCL)* in Listing 5.7; however in this case we check also the running system. Listing 5.13 shows the java code realizing the query.

Listing 5.13: JARQuery to detect missing MimeType handlers

```
...
public class MissingMimeTypeHandler {

  static  Configuration configuration = null;

  public static void checkMissingMimeTypeHandlers() {

    // Checking for missing files
    System.out.println("Cheking for missing MimeTypeHandlers....");

    Iterator<MimeTypeHandler> mthit = configuration.getEnvironment().getMimeTypeHandlerCache().getHandlers().iterator();
    MimeTypeHandler mth = null;
    java.io.File aux = null;

    while ( mthit.hasNext() ) {
      mth = mthit.next();
      aux = new java.io.File(mth.getHandler().getLocation());
      if ( ! aux.exists() ) {
        System.out.println("WARNING: L'handler " +mth.getHandler().getLocation() + " is missing and it is necessary to manage the "  + mth.getType().getName() + " MimeType");
    }

```

```
  }
  System.out.println("Cheking␣for␣missing␣MimeTypeHandlers....DONE");
}


  public static void main(String[] args) throws IOException{
    MancoosiPackageImpl.init();

    URI fileURI = URI.createFileURI("model/systemModel.ecore");
    Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);
    resource.load(null);
    UbuntuConfigurationManager.getInstance().setConfiguration( (Configuration) resource.
        ↪getContents().get(0) );
    configuration = UbuntuConfigurationManager.getInstance().getConfiguration();

    checkMissingMimeTypeHandlers();
  }
}
```

The solution associated to this query suggests to restore the configuration file.

Missing services

This query analyses the running system to check if, according to the configuration model, service binaries are missing. This query is similar to the query *Missing executable in a Service* in Listing 5.9. Listing 5.14 shows the java code realizing the query.

Listing 5.14: JARQuery to detect missing services

```
...
public class MissingServices {

  static  Configuration configuration = null;

  public static void checkMissingServices() {

    System.out.println("Cheking␣for␣missing␣Services....");
    Iterator<Service> is = configuration.getServices();

    while (is.hasNext()){
      Service s = is.next();
      java.io.File aux = new java.io.File(s.getExecutable().getLocation() );
      if ( ! aux.exists() ) {
        System.out.println("WARNING:␣The␣binary␣" + aux.getPath()  + "␣of␣the␣service␣"
            ↪+ s.getName() + "is␣missing.");
      }
    }
    System.out.println("Cheking␣for␣missing␣Services....DONE");
  }


  public static void main(String[] args) throws IOException{
    MancoosiPackageImpl.init();

    URI fileURI = URI.createFileURI("model/systemModel.ecore");
    Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);
    resource.load(null);
    UbuntuConfigurationManager.getInstance().setConfiguration( (Configuration) resource.
        ↪getContents().get(0) );
    configuration = UbuntuConfigurationManager.getInstance().getConfiguration();

    checkMissingServices();
  }
}
```

The solution associated to this query suggests to reinstall the service.

Summing up

Table 5.1 summarizes the case study subjects described above. In particular, Table 5.1 shows the query name, the link to the associated listing, the type of query, i.e, OCL or JAR, and if a solution to the detected fault is provided.

	Query	Listing ref.	Query type	Provide Solution
1	Missing packages involved in implicit package dependencies	Listing 5.2	OCL	Yes
2	Existence of packages that are in the `Half-installed` state	Listing 5.3	OCL	Yes
3	Existence of packages that are in the `Half-configured` state	Listing 5.4	OCL	Yes
4	Existence of packages that are in the `Half-installed` state with a `Reinst requ.` flag	Listing 5.5	OCL	Yes
5	Existence of packages that are in the `Half-configured` state with a `Reinst requ.` flag	Listing 5.6	OCL	Yes
6	Missing executable in a MIME type handler specification (OCL)	Listing 5.7	OCL	Yes
7	Missing executable in a menu entry	Listing 5.8	OCL	Yes
8	Missing link in a Service	Listing 5.9	OCL	Yes
9	Missing link in an Alternative	Listing 5.10	OCL	Yes
10	Existence of conf. files which can be written and/or executed by a not root user	Listing 5.11	OCL	Yes
11	Missing configuration files	Listing 5.12	JAR	Yes
12	Missing executable in a MIME type handler specification	Listing 5.13	JAR	Yes
13	Missing services	Listing 5.14	JAR	Yes

Table 5.1: Case study subjects

5.2.2 Experimental setup

The experiment has been performed on an Intel Core2 Duo CPU P8600 - 2.40 GHz, 4 GB of RAM, with installed a Debian Squeeze distribution. The considered system consists of 1'296 installed packages, and the size of the system configuration model is of ≈3MB.

5.2.3 Results

To validate the fault detector we intentionally injected errors in the real system and we used the fault detector to detect them. More specifically, for each query we injected 20 errors and the fault detector discovered all of them. For instance, referring to the query *Missing configuration files* - Listing 5.12, we intentionally deleted some configuration files of installed packages. The query was able to check both the model and the running system and inform the user about such missing

files. It is important to note that performing the same query without a uniform representation of different configuration aspects requires the adoption of different tools like `dpkg`, `find`, `awk`, and `sed`. For instance, in order to retrieve the configuration files of an installed package the user has to consider the package metadata. Contrariwise, such information is explicitly represented in the system configuration model and immediately available.

Table 5.2 reports the execution time of the case study subjects of Table 5.1. It is important to note that execution time of a query is affected by other aspects which are not directly related to the query execution (e.g., model loading). In fact, executing one of the considered query does not require much more time then executing all of them together (see the last 3 lines of Table 5.2).

Case study subject	Execution time
1	1.487s
2	1.512s
3	1.492s
4	1.452s
5	1.496s
6	1.544s
7	1.476s
8	1.584s
9	1.380s
10	1.504s
11	1.276s
12	1.245s
13	1.124s
1-10	1.660s
11-13	2.204s
1-13	3.296s

Table 5.2: Fault detector experiment results

5.2.4 Threats to Validity

The results of our experiments are subjects to the following threats of validity:

- We considered a real distribution; while this is a large case study, we cannot claim that the results of our experimental evaluation are generalizable. The evaluation should be considered as investigating the potential of the technique rather than providing a statement of effectiveness. However, the performed validation provides us an acceptable confidence of the accuracy and correctness of the fault detector.
- The entire EVOSS approach and then also its fault detector component are engineered to be easy generalizable to other distributions. In fact, we instantiated the approach also to the Caixa Magíca RPM distribution[3].
- The tools we have used or implemented could be defective. To counter this threat, we have run several manual assessments and counter-checks.

[3] `http://www.caixamagica.pt`

- We also evaluated the performance of the fault detector and we show in Table 5.2 a summary of the results. This aspect is important for its use in practice.

Chapter 6

Related Work

In this chapter we compare Evoss and its components with related work. More specifically, Section 6.1 compares Evoss with current meta-installers. Section 6.2 discusses simulation in computer science and how the Evoss simulator is an advance with respect to state of art approaches. Section 6.3 presents approaches to manage safe upgrades. Section 6.4 shows how Evoss can be considered an approach to use model at run-time. Finally, Section 6.5 compared the fault detector with lightweight formal methods.

6.1 Comparing Evoss with current meta-installers

Current meta-installers use algorithms to detect if a solution to the satisfiability problem of package dependencies and conflicts exists and then generate an upgrade plan.

Then meta-installers act as dumb-agents and call `dpkg`, `rpm` or the associated installer on the packages and report the error codes at this stage. Then, the installer unpacks the files downloaded by the meta-installer and runs the associated configuration scripts. It is at this stage that the meta-installers assume that the configuration files will report an error code if something has gone wrong.

It might be the case that the package maintainer may have made a mistake, not made a reference to a dependency or some other types of error within the maintainer file. If this is the case then the maintainer scripts will still run. They are after all written in Turing complete languages. Although they may be syntactically correct they may not be logically correct. There is a large assumption made at this point in that the maintainer is expected to write conditional checks against anything that might fail. Utilities such as deb-helper can help producing maintainer scripts for packagers but it does not guarantee that they will be free of configuration errors. Most of the time when a failure is detected by a meta-installer it will report the error code and stop the installation. The packages that have been installed to this point are normally left as they are and all the commands that were run up to the point of the failure are left as they were. In this case it leaves the system in an undefined state. Files may have been modified and cache-updaters may have been run but the system currently will just say that the package installation failed. This leaves the possibility that some packages were installed, being dependencies of the desired package but are not being used. This entails all the standard problems of failed installations.

There are possible security holes as the installation will be reported back as not have being

	Existing approaches	Model driven approach
Static deploy-time failures	Y	Y
Dynamic deploy-time failures	Y[1]	Y
Undetected failures	N	Y

Table 6.1: Current support to *detect* and *manage* upgrade failures

successful. Also there may be resources that have been left in an erroneous state. For instance, if there is a power failure after a pre-rm script is run, then the caches may have been updated to report that the application has been removed when it has not. Having a transactional system that monitors the configuration states will highlight these errors and possibly revert all the changes dependent on the user's preferences.

To summarize, Table 6.1 reports the main categories of possible upgrade failures that we identified and described in Section 2.2. For each of them the table states if current approaches are able or not to *detect* and eventually *manage* them. EVOSS has been conceived to focus on detecting faults before, during and after upgrades are performed.

6.2 Simulation in computer science

Simulation in computer science is a quite old and established practice (see for instance the *Simulation Modelling Practice and Theory* Elsevier journal[2]). Smith in [Smi00] defines simulation as "*... the process of designing a model of a real or imagined system and conducting experiments with this model to understand the behavior of the system or to evaluate strategies for its operation.*". The key issues in simulation are *model design*, i.e., the acquisition of valid source information about the relevant selection of key characteristics and behaviours, *model execution*, i.e., the use of simplifying approximations and assumptions within the simulation, and *execution analysis*, i.e., fidelity and validity of the simulation outcomes [Fis95].

Typically, simulators are defined and specialized for the specific domain they want to deal with, such as marine simulators, automobile simulators, robotics simulators, city and urban simulators, etc. In this document we proposed a simulator that is specialized to the domain of Linux distributions. The simulator is part of the EVOSS approach and in particular uses the EVOSS DSL for specifying maintainer scripts. EVOSS metamodels are used to represent the system configurations and packages to be installed. They are defined by following a process that makes use of a domain analysis performed to understand details of the domain [DCDRP+10], and is iterative so to tune the model to unforeseen aspects that have to be considered for the simulation purposes.

The importance of the validity of the simulation in terms of fitness for purpose is emphasized in [PAG+10] where the authors show how essential is a trusting relationship between scientists and simulation developers for scientifically-useful and acceptable simulation in scientific domains. Even the Sargent's model of simulation development process [Sar86] gives emphasis on validation of models with respect to the real problem. Moreover, Sargent remind us in [Sar98]

[1] Failures may be detected as the script will report an error code but the changes already performed are usually not undone.

[2] `http://www.sciencedirect.com/science/journal/1569190X`, formerly known as *Simulation Practice and Theory*.

that "*a model should be developed for a scientific purpose ... and its validity determined with respect to that purpose*". We espouse these conclusions and we investigate within Evoss the use of model-driven techniques as a means for realizing 'trusted' simulations. More specifically, we automatize the construction of the system model on which the simulation is performed. In this way we encode the trusting relationship within the injectors implementation in the sense that it is sufficient to test the injectors for trusting on the built model. The injectors in fact build models that conform to metamodels that are a formalization of the domain. Metamodels are defined through an iterative process that involves also domain experts. In other words the metamodels define the abstraction level of the models and embed the 'purpose' of the model according to [Sar98]. Finally, it is important to note that when the injectors have been built they can be executed even before each simulation, as we do in Evoss. This means that the models will be in some sense available@run-time (see Section 6.4) and this would help on maintaining the 'trust' [PAG+10] on the models used for the simulation. Another important aspect is the trust of the simulator itself. This aspect will be discussed in the next chapter.

6.3 Managing safe upgrades

The challenge of the configuration management is on maintaining and deploying the configuration of machines over time. Several approaches and Configuration Management Tools (CMTs) exist to that end; the most relevant ones are Bcfg2 [DBL06], LCFG [AS00], PoDIM [DJ07], Quattor [CPL+08], and Puppet [Kan06, Kan03]. The main difference of our work with respect to the configuration management research 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. On the contrary, the main activity of the Evoss simulator is about simulating the execution of maintainer scripts defined in terms of the Evoss DSL.

Conary [Joh05] 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 [McQ05] 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. This 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[3] 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.

[3] `http://nsis.sourceforge.net/Main_Page`

Finally, NixOS [DL08] 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).

6.4 Models@runtime

The models@runtime approach [BBF09] aims at extending the applicability of models to the run-time environment. Modeling techniques provide viable means to enable system monitoring, model analysis and evolution at run-time [GC08]. These approaches recognize the need to produce, manage, and maintain software models all along the software's life time in order to assist the realization and validation of system's evolution while the system is in execution. The DiVA project[4] combines aspect-oriented and model-driven techniques with the aim to provide a tool-supported methodology and an integrated framework for managing dynamic variability in adaptive systems. Evoss [DCDRP+10] uses models@runtime to manage the upgrade of system configurations in the context of free and open source software systems.

6.5 Lightweight formal methods

The literature is plenty of formal methods [CW96, Pel01], i.e., of techniques that are used to model complex systems as mathematical entities, and that aim at improving the reliability of systems. Examples for formal methods are model checking [CGP01, PPS09], which uses some software to automatically check that the software satisfies its specification, deductive verification [Flo67, Hoa83], which uses some logical formalism to formally prove that the software satisfies its specification, and testing [MS04], which checks software executions according to some coverage scheme.

A particular class of formal methods is the class of lightweight formal methods [JW96]. This kind of formal methods promises to offer a cost-effective and pragmatic way of improving the quality of software specifications.

The main characteristics of lightweight approaches are [JW96]:

- *Partiality in language.* An important aspect of specification languages is their tractability: a specification language intended as a general mathematical notation come at the expense of analysis.
- *Partiality in modelling.* Focus on some aspects of the system, as required by the verification and by keeping low the cost of formalization.
- *Partial analysis.* It would be preferable to reduce the ability to find proofs than the ability to detect errors reliably (a full specification might be undecidable).
- *Partiality in composition.* For a large system, it can be necessary to make use of many partial specifications and views. The composition of the different views is difficult.

[4] `http://www.ict-diva.eu`

The expressiveness of a lightweight formal specification language should be limited to what can be exploited by the automated verification. Automation is extremely important and can be obtained by reducing the descriptive power.

Examples of lightweight formal methods are Alloy [Jac02], Eiffel [Mey91][5], IFAD VDM [AL99], and MOSEL-2 [Bar05].

Alloy is a small modelling language for describing structural properties. It is easy to read and write, and provides fully-automated analysis.

Eiffel is a comprehensive framework for developing high quality software at low cost. The reliable design and code is obtained through the use of the *design by contract* method [Mey98]. Design by contract is a software development method that promotes the design of a system in terms of components that cooperate on the basis of precisely defined contracts.

The Vienna Development Method (VDM) is a formal method that supports the specification and design of software systems. It supports syntax checking, extensive static semantics checking, and documentation.

MOSEL-2 is a lightweight formal method approach for performance evaluation of real-world systems whose dynamic behaviour can be characterized as a discrete event triggered evolution.

The failure detector is a sort of lightweight formal method. It is fully automatic, focused in the specific domain, and effective. We use OCL instead of using an existent lightweight approach, such as Alloy, since OCL can be directly used with any Meta-Object Facility (MOF) meta-model.

[5] `http://www.eiffel.com/`

Chapter 7

Conclusion

In this deliverable 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 three widely used FOSS distributions, namely Caixa Magíca, Debian, and Ubuntu. 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.

In the following we provide some considerations on Evoss, revolving around three topics: community support, flexibility, generalization, and performance of Evoss.

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 faults is relatively straightforward in Evoss: it is enough to add the corresponding OCL or JAR queries to the fault 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 [CPL+08]) 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 [DPP10] (and recalled 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 fault 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 3.5) 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.

Generalization of the simulator The overall EVOSS approach can be generalized and the lessons learned in our experience might be reused even in other contexts different from FOSS systems.

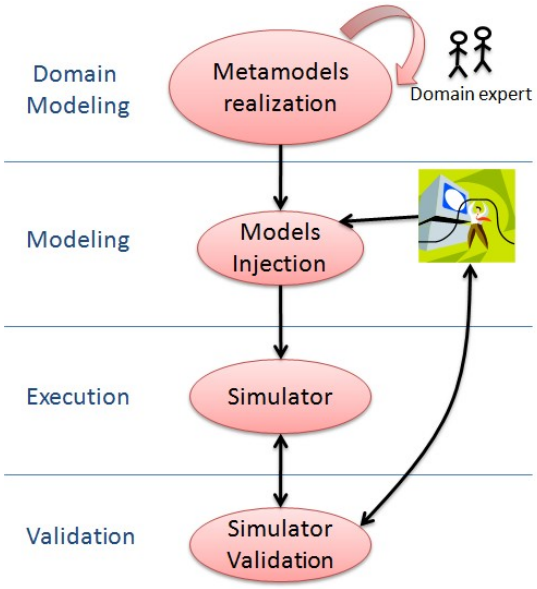


Figure 7.1: Model-driven simulation approach

Our thesis is that model-driven technologies might help the realization of 'trusted' simulations, by espousing the terminology in [PAG+10]. Figure 7.1 shows the phases that compose the model-driven simulation approach. An analysis and representation of the domain of interest is unavoidable. In our approach we realize and encode the domain modeling in term of metamodels. The process of realizing the metamodels, which involves also the identification of elements to be considered, is naturally iterative and should include domain experts: they can provide the needed knowledge and simulator developers have to build a trust relationship

with them [PAG+10]. The realization of metamodels enables the automatic conformity check of models on which the simulation is performed. This conformity ensures that the produced models will respect constraints defined at the metamodel level and that the models will be produced at the 'correct' abstraction level, i.e., as encoded in the metamodels.

Another important aspect is that models can be automatically produced thanks to model injection techniques, such as [JJJ08] and the MoDisco project[1], which are tailored on extracting models from source code. Specifically, the package injection component of Evoss uses Gra2Mol [JJJ08] to automatically extract models from maintainer scripts written in Bash. Moreover, the configuration injection component automatically build a configuration model by querying the running system, by extracting useful information, and by programmatically building models. This has several advantages:

- models are automatically built, then once we are confident on the injector correctness we can trust the produced models;
- since the construction of the models is automatized, we can re-perform the model construction before each simulation; this ensures maintainability also of the trust on the models [PAG+10].

The simulation is performed by executing chains of model transformations that starting from a system configuration produce a new system configuration. Model transformations enable also the construction of error models containing the reason of the error.

Even the validation of the simulator can profit from the model-driven infrastructure. In addition to the fact that the construction of the models is automatized by using model injectors that can be largely tested as any other software program, specific validation procedures can be defined. For instance, we realized an automatic validation check, as described in Section 4.6, which increases the confidence on the simulator. Specifically, starting from a system configuration CM_1 we inject a configuration model and we perform the simulation, thus producing a new configuration model CM_2. Then we perform the real system upgrade and we inject again the system, thus producing a configuration model CM_3. We are then able to check that CM_2 is equal to CM_3. By testing this on a number of system upgrades we validate the simulator in terms of fitness for purpose.

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 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 $\approx 1 \cdot 400$ installed packages. The whole injection process and the execution of the failure detector described in Chapter 5, have been completed in less than 240 seconds, as shown in Figure 7.2. Another experiment has been performed on an Ubuntu 9.10 system, running on a virtual machine with 512MB of RAM upon Virtual Box[2]. In this case the process took longer due to the limited hardware resources. It has been completed nevertheless in $\approx$1000 seconds.

[1] `http://www.eclipse.org/MoDisco/`
[2] `http://www.virtualbox.org/`.

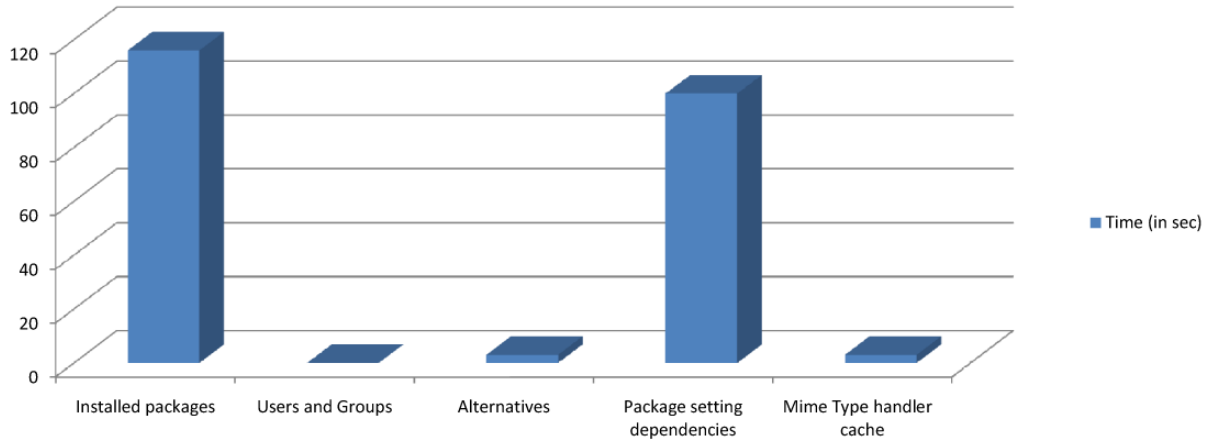


Figure 7.2: Timing injection of an Ubuntu 9.10 system on a test machine

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.

Bibliography

[ACTZ11] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Mpm : a modular package manager. In *CBSE 2011*, June 21-23, 2011.

[AL99] Sten Agerholm and Peter Gorm Larsen. A lightweight approach to formal methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 168–183, London, UK, 1999. Springer-Verlag.

[Apa] The Apache Ant Project, http://ant.apache.org/. *Apache Ant Homepage*.

[AS00] Paul Anderson and Alastair Scobie. Large scale Linux configuration with LCFG. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, pages 42–42. USENIX, 2000.

[BÓ5] J. Bézivin. On the Unification Power of Models. *SOSYM*, 4(2):171–188, 2005.

[Bar05] Jörg Barner. *A Lightweight Formal Method for the Prediction of NonFunctional System Properties*. PhD thesis, Der Technischen Fakultät der Universität Erlangen-Nürnberg, 2005.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. Models run.time. *Computer*, 42:22–27, 2009.

[BIPT09] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[BW98] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[CC77] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. *SIGSOFT Softw. Eng. Notes*, 2:77–94, March 1977.

[CDP07] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron. A. Peled. *Model Checking*. The MIT Press, Massachusetts Institute of Technology, 2001.

[CKK+07] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev.*, 41(6):221–236, 2007.

[CPL+08] Stephen Childs, Marco Emilio Poleggi, Charles Loomis, Luis Fernando Muñoz Mejías, Michel Jouvin, Ronald Starink, Stijn De Weirdt, and Germán Cancio Meliá. Devolved management of distributed infrastructures with Quattor. In *LISA*, pages 175–189. USENIX Association, 2008.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28:626–643, December 1996.

[DBL06] Narayan Desai, Rick Bradshaw, and Cory Lueninghoener. Directing change using Bcfg2. In *LISA*, pages 215–220. USENIX, 2006.

[DCDRP+10] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Supporting software evolution in component-based foss systems. *Science of Computer Programming*, (Available online at: http://dx.doi.org/10.1016/j.scico.2010.11.001), 2010.

[DJ07] Thomas Delaet and Wouter Joosen. PoDIM: A language for high-level configuration management. In *LISA*, pages 261–273. USENIX, 2007.

[DL08] Eelco Dolstra and Andres Löh. NixOS: A purely functional linux distribution. In *ICFP*, 2008. To appear.

[DP11] Davide Di Ruscio and Patrizio Pelliccione. Model-driven approach to detect faults in evolving foss systems. *Submitted for publication*, (Technical Report TRCS 003/2011, University of L'Aquila), 2011.

[DPD11] Davide Di Ruscio, Patrizio Pelliccione, and Massimo Del Rosso. Simulating upgrades of component-based foss systems. *Submitted for publication*, (Technical Report TRCS 002/2011, University of L'Aquila), 2011.

[DPP10] Davide Di Ruscio, Patrizio Pelliccione, and Alfonso Pierantonio. Instantiation of the metamodel on a widely used gnu/linux distribution. Mancoosi Project deliverable D2.2, January 2010. `http://www.mancoosi.org/reports/d2.2.pdf`.

[DPPZ09] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Metamodel for describing system structure and state. Mancoosi Project deliverable D2.1, January 2009. `http://www.mancoosi.org/reports/d2.1.pdf`.

[DRPPZ09] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Towards maintainer script modernization in FOSS distributions. In *IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems*, pages 11–20, New York, NY, USA, 2009. ACM.

[DTPP09] Davide Di Ruscio, John Thomson, Patrizio Pelliccione, and Alfonso Pierantonio. First version of the DSL based on the model developed in WP2. Mancoosi Project deliverable D3.2, November 2009. `http://www.mancoosi.org/reports/d3.2.pdf`.

[DTZ08] Roberto Di Cosmo, Paulo Trezentos, and Stefano Zacchiroli. Package upgrades in FOSS distributions: Details and challenges. In *HotSWup'08*, 2008. To appear.

[DW99] D. D'Souza and A.C. Wills. *Objects, components, and frameworks with UML: The catalysis approach*. Addison-Wesley, Boston, MA, 1999.

[Fis95] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1995.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[GC08] Heather J. Goldsby and Betty H. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 568–583, Berlin, Heidelberg, 2008. Springer-Verlag.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GRM+09] Jesus Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan Amor, and Daniel German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26:53–56, January 1983.

[IC02] et al. I. Crnkovic. Anatomy of a research project in predictable assembly. In *In 5th ICSE Workshop on Component Based Software Engineering*. ACM, May, 2002.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.

[JJJ08] J.L.C. Izquierdo, J.S. Cuadrado, and J.G. Molina. Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization. In *Workshop on Model-Driven Software Evolution*, 2008.

[Joh05] Michael K. Johnson. Building linux software with conary. In *Proceedings of the Linux Symposium*, 2005.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, 1996.

[Kan03] Luke Kanies. ISconf: Theory, practice, and beyond. In *USENIX-LISA'03*, pages 115–124. USENIX Association, 2003.

[Kan06] Luke Kanies. Puppet: Next-generation configuration management. *the USENIX magazine*, 31(1):19–25, 2006.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[LR00] M. M. Lehman and Juan F. Ramil. Software evolution in the age of component-based software engineering. *IEE Proceedings - Software*, 147(6):249–255, 2000.

[MBC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, pages 199–208, Tokyo, Japan, September 2006. IEEE CS Press.

[McQ05] Robert McQueen. Creating, reverting & manipulating filesystem changesets on Linux. Part II Dissertation, Computer Laboratory, University of Cambridge, May 2005.

[MDe08] Tom Mens, Serge Demeyer, and (eds.). *Software evolution*. Springer, 2008.

[Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[Mey98] Bertrand Meyer. Design by contract: The eiffel method. In *TOOLS (26)*, page 446, 1998.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *IWPSE '05*, 2005.

[NR68] P. Naur and B. Randell. Software engineering report on a conference sponsored by the NATO science committee. page 231, October 1968.

[PAG+10] Fiona A. C. Polack, Paul S. Andrews, Teodor Ghetiu, Mark Read, Susan Stepney, Jon Timmis, and Adam T. Sampson. Reflections on the simulation of complex systems for science. In *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 276–285. IEEE Press, March 2010.

[Pel01] Doron A. Peled, editor. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[PPS08] Doron Peled, Patrizio Pelliccione, and Paola Spoletini. Model checking. In *Wiley Encyclopedia of Computer Science and Engineering*. 2008.

[PPS09] Doron Peled, Patrizio Pelliccione, and Paola Spoletini. *Wiley Encyclopedia of Computer Science and Engineering, 5-Volume Set*, chapter Model Checking. Wiley, 2009.

[Ray01] Eric S. Raymond. *The cathedral and the bazaar*. O'Reilly, 2001.

[RRGLR+09] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL model transformations. In *Proc. of MtATL 2009*, pages 34–46, Nantes, France, July 2009.

[Sar86] Robert G. Sargent. The use of graphical models in model validation. In *Proceedings of the 18th conference on Winter simulation*, WSC '86, pages 237–241, New York, NY, USA, 1986. ACM.

[Sar98] Robert G. Sargent. Verification and validation of simulation models. In *Proceedings of the 30th conference on Winter simulation*, WSC '98, pages 121–130, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Smi00] Roger D. Smith. Simulation. In *Encyclopedia of Computer Science*, pages 1578–1587. John Wiley and Sons Ltd., Chichester, UK, 2000.

[TZ09] Ralf Treinen and Stefano Zacchiroli. Expressing advanced user preferences in componet installation. In *IWOCE'09*, pages 31–40. ACM, 2009.

[UBC09] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.*, 35:384–406, May 2009.

[Vig01] Mark Vigder. *The Evolution, Maintenance and Management of Component-Based Systems*. Addison-Wesley, 2001.

[VR02] Marlon Vieira and Debra Richardson. The role of dependencies in component-based systems evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE '02, pages 62–65, New York, NY, USA, 2002. ACM.

[WLC01] Brian Witten, Carl E. Landwehr, and Michael A. Caloyannides. Does open source improve system security? *IEEE Software*, 18(5):57–61, 2001.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*, pages 179–192, 2006.