mancoosi
managing software complexity

Instantiation of the metamodel on a widely used GNU/Linux distribution

Deliverable 2.2

Nature : Deliverable
Due date : 31.01.2010
Start date of project : 01.01.2008
Duration : 36 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>
Internal review	Stefano Zacchiroli <zack@pps.jussieu.fr>
Workpackage number	WP2
Deliverable number	2
Document type	Deliverable
Version	1
Due date	31/01/2010
Actual submission date	31/01/2010
Distribution	Public
Project coordinator	Roberto Di Cosmo <roberto@dicosmo.org>

Abstract

One the main objectives of the Mancoosi project is to propose a model-driven approach to improve the upgrade of FOSS installations. Equipped with that, package managers can both simulate upgrades (trying to detect configuration inconsistencies) and, during deployment on the real system, create a more detailed log of script executions that can be used later on to pinpoint upgrade roll-back mechanism to the precise point where the failure occurred during deployment.

In order to enable the Mancoosi model-driven approach, this deliverable presents the Mancoosi model injection technique that is devoted to automatically extract models from a running Linux system. The model injection approach is implemented in Java and makes use of the Eclipse Modeling Framework (EMF) facilities. The approach has a layered structure: only the more specialized layer, which is in charge of executing shell commands, is specifically defined for the considered distribution and need to be replicated for each new different distribution. The other layers create models starting from the retrieved information in a way that is independent by the considered distribution. The result of the Mancoosi injection approach is a model of the running system conforming to the metamodel presented in Deliverable D2.1.

In this work we present also techniques for keeping the real system and the models always synchronized. It is in fact important to note that it is always possible that the model and real system become slightly out-of-sync. For instance we cannot forbid a user to manually delete a file or to manually change configuration aspects. For this reason we provide mechanisms for checking the conformity between system and models. Since these synchronization's features are really fast, the synchronization can be performed even before each system upgrade and then before each upgrade simulation.

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

Free and Open Source Software (FOSS) distributions are among the most complex software systems known, being made of tens of thousands of components evolving rapidly without centralized coordination. Similarly to other software distribution infrastructures, FOSS components are provided in "packaged" form by distribution editors. Packages define the granularity at which components are managed (installed, removed, upgraded to newer version, etc.) using *package manager* applications, such as `APT` [Nor08] or `Apache maven` [Mav08]. Furthermore, the system openness affords an anarchic array of dependency modalities between the adopted packages.

One of the main goals of Mancoosi[1] is to define a *model-driven* approach [Bó5] to support the upgrades of FOSS distributions. The main idea of the proposed model-driven approach [CRP+09, CDRP+10, DPPZ09] is to specify in terms of models both system configurations and available packages. In previous works we presented the Mancoosi metamodels (see Deliverable D2.1 [DPPZ09]) and the Domain Specific Language (DSL) that we defined to specify the maintainer scripts (see Deliverable D3.2 [DTP+09]) which have to be executed during package installations and/or upgrades. Intuitively, models are an abstraction of the real system since they focus on the relevant aspects in order to predict the operation effects on the software distribution. To this end, models can be used to drive roll-back operations to recover previous configurations according to user decisions or after upgrade failures. The use of models allow us to programmatically reason about the system and promotes also the definition of constraints and rules that enable some significant static analysis of system configurations. Moreover, the use of models enables the simulation of the system upgrade. In other words, an upgrade can be simulated on the models before performing the real upgrade of the system. Since the models contain also the description of the scripts, in terms of DSL statements, the simulation is not only a resolution of explicit dependencies among packages, but it deals also with implicit dependencies and with failures that can be caused by unavailable resources.

In this deliverable we present the instantiation of the Mancoosi metamodel on a Debian-based distribution. In particular, we describe both system configuration and package maintainer scripts in terms of models. Since the dimension of the considered system is sizeable, we propose an approach to support the automatic generation of system configuration and package models. More precisely, in this deliverable we present a model injection approach [Ecl] that is devoted to automatically extract models from real artifacts [RPPZ09]. The idea of renewing legacy systems by means of model driven approaches has been pursued by the Object Management Group

[1] Mancoosi project: `http://www.mancoosi.org`

(OMG) since 2003. In particular, OMG defined the Architecture-Driven Modernization (ADM) task force [KU07] to support software modernization of existing assets which are imported into MDE enabled development environments. The developed systems have been designed in a modular way, thus the support for new distributions can be easily added by extending and using the provided infrastructure.

In the Mancoosi context we focus on the automatically extraction of models from a real system that is running on given machines. When dealing with models, it is always an open question how to ensure both that the models expose the right abstraction level, i.e. they contain all the useful information, and therefore the models accurately represent the reality. In our approach the abstraction level is fixed by the Mancoosi metamodels obtained though a suitable domain analysis described in Deliverable D2.1. The confidence about the fact that our models accurately represent the real system is gained by the fact that we programmatically and automatically build our models. In other words the check about the correctness of the models is moved from the analysis of the models themselves to the automatism defined for obtaining the models. Therefore, the accuracy of the models can be easily checked by suitably testing the defined automatism.

Once we are confident that the models accurately represent the real system another problem raises up. Are we sure that operating with the real system and performing upgrades system and models will be kept aligned? One first aspect to consider is that for each upgrade the models are consistently updated. However, it is always possible that the model and real system become slightly out-of-synch. We cannot in fact forbid a user to manually delete a file or to manually change configuration aspects. For this reason we provide mechanisms for checking the conformity between system and models. Since these synchronization's features are really fast, the synchronization can be performed even before each system upgrade and then before each upgrade simulation.

The model injection is implemented in Java and makes use of the Eclipse Modeling Framework (EMF)[2]. The model injection is split in two parts: the *system configuration injection* which deals with the injection of the static part of the system, e.g. file system, packages, mime-types, and the *package injection* which deals with the injection of the dynamic part, i.e. maintainer scripts. The package injection will be a fundamental part of the upgrade simulation, which will be the objective of Deliverable D2.3, and of the support for the run-time system upgrade which will be described in Deliverable D3.3. Both system configuration and package injections have a layered structure which has been defined in order to have a distribution-independent injection. In fact only the more specialized layer is specific for the considered distribution and need to be replicated for a different distribution. This is unavoidable since this layer needs to execute specific shell commands to retrieve system and packages information. Contrariwise, the other layers create models starting from the retrieved information in a way that is independent by the considered distribution. The created models conform to the metamodels presented in Deliverable D2.1.

1.1 Structure of the deliverable

This deliverable is structured in six chapters:

- Chapter 1 contains an outline of the Deliverable and discusses the context in which this work appears;

[2]Eclipse Modeling Framework (EMF) project : http://www.eclipse.org/emf

- Chapter 2 briefly recalls FOSS distributions, the Mancoosi model-driven approach, the Mancoosi metamodels and finally provides an overview of the model injection problem;
- Chapter 3 describes the Mancoosi model configuration injection, by describing the used technologies and the different layers that compose the model injection approach;
- Chapter 4 shows the instantiation of the overall model injection on a real system. More precisely we report our experience injecting a running system with the Ubuntu 9.10 distribution installed;
- Chapter 5 describes the Mancoosi package injection fundamental part for performing the upgrade simulation. This injection in fact deals with the injection of maintainer scripts that are executed during the system's upgrade;
- Chapter 6 shows some static analysis that we can perform on the generated models;
- Chapter 7 concludes the deliverable and outlines future research directions.

Chapter 2

The model driven approach to support the upgrade of FOSS systems

This section recalls the Mancoosi model-driven approach with the aim to show how the work presented in this deliverable can be integrated with the overall Mancoosi approach. Section 2.1 provides an overview of the FOSS distributions and of their packaged nature. Section 2.2 outlines the overall Mancoosi approach. Section 2.3 recalls the Mancoosi metamodels presented in Deliverable D2.1 and finally, Section 2.4 shows the role that the injectors have in the Mancoosi model-driven approach.

2.1 FOSS distributions

Free and Open Source Software (FOSS) distributions are among the most complex software systems known, being made of tens of thousands components evolving rapidly without centralized coordination. Similarly to other software distribution infrastructures, FOSS components are provided in "packaged" form by distribution editors. Packages define the granularity at which components are managed (installed, removed, upgraded to newer version, etc.) using *package manager* applications, such as: *APT* [Nor08], Smart [Nie08], Apache Maven [Mav08].

Overall, the architectures of all FOSS distributions are quite similar. Each user machine, i.e. a distribution *installation*, has a local *package status* recording which packages are locally installed and which are available from remote distribution repositories. In an *upgrade scenario* the system administrator requests a change of the package status (e.g. install, remove, upgrade to a newer version) by means of a package manager, which is in charge of finding a suitable *upgrade plan*. More precisely, the package manager solves dependencies and conflicts, retrieves packages from remote repositories as needed, and deploys individual packages on the filesystem, possibly aborting the operation if problems are encountered along the way.

A *package* is usually a bundle of three main parts:

- **Files :** this part describes the set of files and directories shipped within the package for installation, e.g. executable binaries, data, documentation, etc. *Configuration files* is the subset of files affecting the runtime behavior of the package and meant to be locally

customized by the system administrator. Proper internalization of configuration file details is relevant for our purposes, as specific configurations can (implicitly) entail dependencies not otherwise declared by the involved packages. Configuration files is one of the meta-information of a package.

- **Meta-information :** this part contains package-related information, such as a unique identifier, software version, maintainer and package description, and most notably *inter-package relationships*. The kinds of relationships vary with the distribution, but a common core subset includes: dependencies (the need of other packages to work properly), conflicts (the inability of being co-installed with other packages), feature provisions (the ability to declare named features as provided by a given package, so that other packages can depend on them), and restricted boolean combinations of them [EDO06], as can be seen in Listing 2.1.

  Example 1 This example shows the meta-information of the package `firefox-3.5` of the Ubuntu 9.10 distribution. As shown in Listing 2.1, there are several information about a package.

Listing 2.1: Package meta-information

```
1  Package: firefox-3.5
2  Status: install ok installed
3  Priority: optional
4  Section: web
5  Installed-Size: 3640
6  Maintainer: Ubuntu Mozilla Team <ubuntu-mozillateam@lists.ubuntu.com>
7  Architecture: i386
8  Version: 3.5.7+nobinonly-0ubuntu0.9.10.1
9  Replaces: firefox-3.0 (<< 3.1~), firefox-3.0-dom-inspector (<< 3.1~), firefox-3.0-
       ↪venkman (<< 3.1~), firefox-3.1, firefox-dom-inspector (<< 3.1~)
10 Provides: firefox-3.0, firefox-3.0-dom-inspector, firefox-3.0-venkman, firefox-3.1,
       ↪ firefox-dom-inspector, www-browser
11 Depends: fontconfig, psmisc, debianutils (>= 1.16), xulrunner-1.9.1 (>= 1.9.1),
       ↪libasound2 (>> 1.0.18), libatk1.0-0 (>= 1.20.0), libc6 (>= 2.4), libcairo2
       ↪(>= 1.2.4), libfontconfig1 (>= 2.4.0), libfreetype6 (>= 2.2.1), libgcc1 (>=
       ↪1:4.1.1), libglib2.0-0 (>= 2.16.0), libgtk2.0-0 (>= 2.10), libnspr4-0d (>=
       ↪4.7.3-0ubuntu1~), libpango1.0-0 (>= 1.14.0), libstdc++6 (>= 4.1.1), firefox
       ↪-3.5-branding | abrowser-3.5-branding
12 Recommends: ubufox
13 Suggests: firefox-3.5-gnome-support (= 3.5.7+nobinonly-0ubuntu0.9.10.1), latex-xft-
       ↪fonts, libthai0
14 Conflicts: firefox-3.0 (<< 3.1~), firefox-3.0-dom-inspector (<< 3.1~), firefox-3.0-
       ↪venkman (<< 3.1~), firefox-3.1 (<< 3.1~b4~hg20090317), firefox-dom-inspector
       ↪(<< 3.1~)
15 Conffiles:
16  /etc/apparmor.d/usr.bin.firefox-3.5 97d80b2693f5e7d9141a4275b91bd883
17  /etc/firefox-3.5/profile/bookmarks.html 111416629615cf48b389d1c5d5c11c27
18  /etc/firefox-3.5/profile/localstore.rdf ea03cc19c2a3f622fa557cd8ea9da6eb
19  /etc/firefox-3.5/profile/prefs.js 99940ecd258d83b3355ab06fca0ffddb
20  /etc/firefox-3.5/profile/mimeTypes.rdf 6047f42624d9930caa8d651fa94d28f1
21  /etc/firefox-3.5/profile/chrome/userChrome-example.css
        ↪c63733eef9d337c86e6609bcc478a668
22  /etc/firefox-3.5/profile/chrome/userContent-example.css
        ↪d3765c7d2de5626529195007f4b7144a
23  /etc/firefox-3.5/pref/firefox.js 7883cf0689295efef7b5b05472e28461
24 Description: safe and easy web browser from Mozilla
25  Firefox delivers safe, easy web browsing. A familiar user interface,
26  enhanced security features including protection from online identity theft,
27  and integrated search let you get the most out of the web.
```

In particular we can see the package name (`Package`) and the version (`Version`) that univocally identify the package. Then we have a set of information that relates the package with other existing packages. There are several kinds of relations among packages. For instance, `Replaces` means that the package substitutes other packages, in this case

previous versions of Firefox. `Depends` represents the dependencies of the package with other packages. `Recommends` is a mild dependency that, if it is not satisfied, this will not cause problems. `Suggests` is just a suggestion for other packages that could be useful. `Conflicts` represents other packages that create conflict with the considered one; the meaning is that a configuration should never have two conflicting packages installed at the same moment. `Provides` represents the definition of a virtual package. The virtual packages only exist logically, not physically. The packages with this particular function will then provide the virtual package. Thus, any other package requiring that function can simply depend on the virtual package without having to specify all possible packages individually. If there are both concrete and virtual packages of the same name, then the dependency may be satisfied (or the conflict caused) by either the concrete package with the name in question or any other concrete package which provides the virtual package with the name in question. Finally, `Description` is a human readable package description, while the field `Conffiles` lists the configuration files of the package.

Configuration files have very different nature and they can collect several kind of information. They could be executable file but they can also contain information that is managed by other programs present in the package. For instance Listing 2.2 shows an html configuration file of the package `firefox-3.5` that is automatically generated. Then this file cannot be manually modified. Contrariwise, Listing 2.3 shows a configuration file that allows users to define specific preferences on Mozilla Firefox.

Example 2 This example shows the the configuration file `/etc/firefox-3.5/profile/bookmarks.html` of the package `firefox-3.5` of the well known browser. This configuration file is automatically generated and it cannot be edited and modified.

Listing 2.2: `/etc/firefox-3.5/profile/bookmarks.html` Configuration file

```
<!DOCTYPE NETSCAPE-Bookmark-file-1> <!-- This is an automatically
generated file.
     It will be read and overwritten.
     DO NOT EDIT! -->
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8">
<TITLE>Bookmarks</TITLE> <H1>Bookmarks</H1>

<DL><p>
    <DT><A HREF="https://addons.mozilla.org/en-US/firefox/bookmarks/" ICON="data:
        ↪image/png;base64,iVBORw0KG...
```

Example 3 This example shows the configuration file `/etc/firefox-3.5/pref/ firefox.js` of the package `firefox-3.5` (see Listing 2.3). This configuration file should be edited in order to select specific preferences, such as enabling/disabling extensions update, enable/disable the checking for default browser and so on.

Listing 2.3: `/etc/firefox-3.5/pref/firefox.js` Configuration file

```
// This is the Debian specific preferences file for Mozilla Firefox
// You can make any change in here, it is the purpose of this file.
// You can, with this file and all files present in the
// /etc/firefox/pref directory, override any preference that is
// present in /usr/lib/firefox/defaults/pref directory.
// While your changes will be kept on upgrade if you modify files in
// /etc/firefox/pref, please note that they won't be kept if you
// do them in /usr/lib/firefox/defaults/pref.

pref("extensions.update.enabled", true);

// Use LANG environment variable to choose locale
pref("intl.locale.matchOS", true);

// Disable default browser checking.
```

```
16 pref("browser.shell.checkDefaultBrowser", false);
17
18 // Prevent EULA dialog to popup on first run
19 pref("browser.EULA.override", true);
20
21 // identify ubuntu @ yahoo searchplugin
22 pref("browser.search.param.yahoo-fr", "ubuntu");
23
24 // identify default locale to use if no /usr/lib/firefox-addons/searchplugins/
       ↪LOCALE
25 // exists for the current used LOCALE
26 pref("distribution.searchplugins.defaultLocale", "en-US");
```

- **Maintainer scripts:** this part defines a set of programs, usually written in shell script, that are used to enable maintainers to attach actions to hooks that are fired by the installer. The set of available hooks depends on the installer; `dpkg` offers one of the most comprehensive set of hooks: pre/post-unpacking, pre/post-removal, and upgrade/downgrade to specific versions [JS08].

  Considering the same example of package, namely `firefox-3.5`, this package has four maintainer scripts: the pre and post installation script, i.e. firefox-3.5.preinst and firefox-3.5.postinst respectively, and the pre and post removal scripts, i.e. firefox-3.5.prerm and firefox-3.5.postrm respectively. The Listing 2.4 shows the firefox-3.5.prerm script.

Listing 2.4: firefox-3.5.prerm Maintainer script

```
1 #!/bin/sh
2
3 set -e
4
5 APPNAME=firefox-3.5
6
7 if [ "1" = "remove" ]||[ "1" = "deconfigure" ] ; then
8     update-alternatives --remove x-www-browser /usr/bin/
          ↪APPNAMErm - f/var/lib/update - notifier/user.d/APPNAME-restart-required
9 fi
10
11 if [ -f /usr/share/apport/package-hooks/
       ↪APPNAME.pyc]; thenrm - f/usr/share/apport/package - hooks/APPNAME.pyc
12 fi
```

Maintainer scripts are challenging objects to model, both for its semantics (shell script is a full-fledged, Turing-complete programming language) and for its syntax which enjoys a plethora of meta-syntactic facilities (here-doc syntax, interpolation, etc.). Moreover they can do anything permitted to the installer, which is usually run with system administrator rights. These characteristics make very hard to predict their, possibly unexpected, side-effects which can span the whole installation.

During package deployment, various kinds of failures can be induced by maintainer scripts. The "simplest" example is a runtime failure of a script (usually detected by a non-zero exit code), against which system administrators are left helpless beside their shell script debugging abilities. More subtle, though possibly easier to deal with, kinds of failures are inconsistent configurations left over by upgrade scenarios not predicted by maintainers. For instance: a maintainer script can "forget" to un-register a plugin from its main application while removing the package shipping the plugin, hence leaving around an inconsistent configuration (which might, or might not, cause execution failures in the main application).

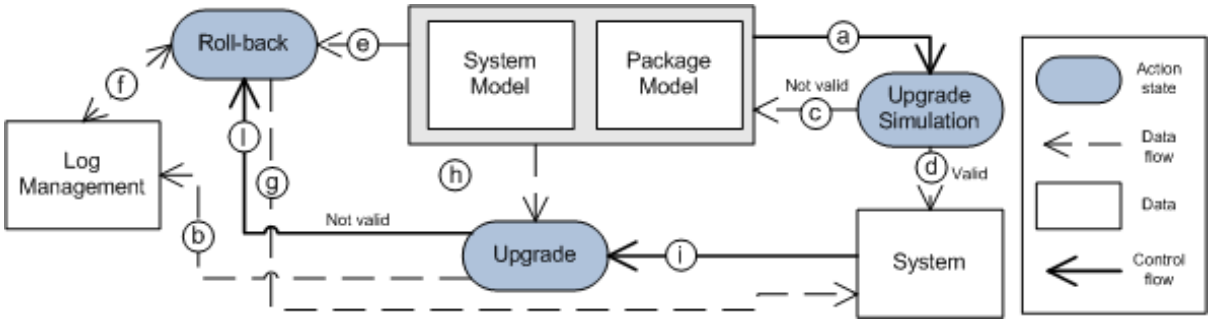

Figure 2.1: Overall approach

2.2 An overview of the model-driven approach

This section briefly recalls the model-driven approach of Mancoosi and we refer to Deliverable D2.1 for an extended discussion of it. Figure 2.1 depicts the Mancoosi model-driven approach. As already highlighted in the introduction, two are the most important features supported by this approach: the simulation and the use of the model at run-time as driver for the down-grade. The simulation takes two models as input: the *System Model* and the *Package Model* (see the arrow ⓐ). The former describes the state of a given system in terms of installed packages, running services, configuration files, etc. The latter provides information about the packages involved in the upgrade, maintainer scripts included. Given a current configuration, defined in terms of system model, the simulation the upgrade of the system is simulated by executing the maintainer scripts of the considered packages. If the execution of a script has some problems or a *not valid* configuration is reached, then the outcome of the simulation will be *not valid* (see the arrow ⓒ). In this case before proceeding with the upgrade on the real system, the problem spotted by the simulation should be fixed. Otherwise, if the overall upgrade leaded to a new configuration that is a *valid* configuration, then the simulation outcome will be *valid* (see the arrow ⓓ). In this case the upgrade on the real system can be operated (see the arrow ⓘ). However, since the models are an abstraction of the reality, upgrade failures might occur.

In order to support the down-grade, during the real upgrade of the system, models are continuously updated and kept aligned with the real system they are modeling. Furthermore, *Log models* are produced in order to store all the transitions between configurations (see arrow ⓑ). The information contained in the system, package, and log models (arrows ⓔ and ⓕ) are used in case of failures (arrow ⓛ) or in case of a user decides to perform a down-grade. In other words, models drive the down-grade, by indicating the actions that must be performed to bring the system back to a previous valid configuration (arrow ⓖ). This model-driven approach, to be effective, must be integrated with other approaches able to physically store useful system information, and able to retrieve these information when needed. For a discussion of this point please refer to Deliverable D3.1.

Since it is not possible to specify in detail every single part of systems and packages, trade-offs between model completeness and usefulness have been evaluated; the result of such a study has been formalized in terms of metamodels which can be considered one of the main constituting concepts of Model Driven Engineering (MDE) [Sch06]. They are the formal definition of well-formed models, constituting the languages by which a given reality can be described in some abstract sense [BÓ5] defining an abstract interpretation of the system. These metamodels are extremely important since they drive the model injection and the models that we obtain as

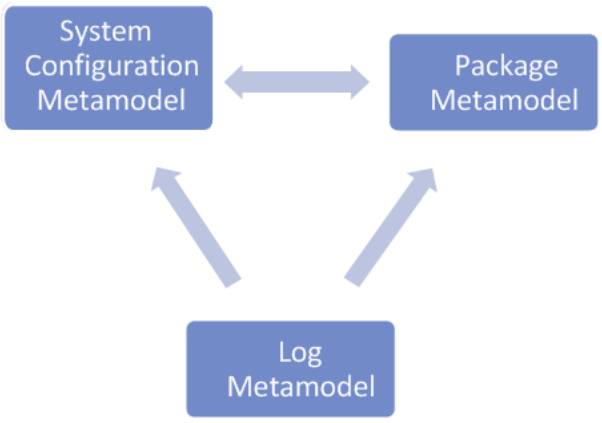

Figure 2.2: Dependencies among metamodels

result of the model injection will conform to these metamodels.

2.3 Mancoosi Metamodels

This section briefly recalls the Mancoosi metamodels while we refer to Deliverable D2.1 for its comprehensive presentation. In order to identify the right trade-off between model completeness and usefulness we analyzed two complex FOSS distributions: the Debian[1], the largest distribution in terms of number of software packages [AGRH05] and the RPM-based Fedora[2] distributions. The analysis can be found on deliverable D2.1 and on [RPPZ09]. Successively the metamodel has been validated by describing part of real systems.

The metamodels which underpin the model based approach are shown in Figure 2.1. The metamodels describe the concepts making up a system configuration and a software package, and how to maintain the log of all upgrades. The metamodels have been defined according to an iterative process consisting of two main steps *a)* elicitation of new concepts from the domain to the metamodel *b)* validation of the formalization of the concepts by describing part of the real systems. In particular, the analysis has been performed considering the official packages released by the distributions with the aim of identifying elements that must be considered as part of the metamodels. Due to space constraints we report here only the results of the analysis, i.e. the metamodels themselves:

- the *System Configuration metamodel*, which contains all the modeling constructs necessary to make the FOSS system able to perform its intended functions. In particular it specifies installed packages, configuration files, services, filesystem state, loaded modules, shared libraries, running processes, etc. The system configuration metamodel takes into account the possible dependency between the configuration of an installed package and other package configurations. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by the proposed metamodels which embody domain concepts which are not taken into account by current package manager tools;
- the *Package metamodel*, which describes the relevant elements making up a software pack-

[1] http://www.debian.org
[2] http://fedoraproject.org

age. The metamodel also gives the possibility to specify the maintainer script behaviors which are currently ignored—beside mere execution—by existing package managers. In order to describe the scripts behavior, the package metamodel contains the `Statement` metaclass that represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration;

– the *Log metamodel*, which is based on the concept of transactions that represent a set of statements that change the system configurations. Transitions can be considered as model transformations [Bó5] which let a configuration C_1 evolve into a configuration C_2.

As depicted in Figure 2.2, *System Configuration* and *Package* metamodels have mutual dependencies, whereas the *Log* metamodel has only direct relations with both *System Configuration* and *Package* metamodels. Figure 2.3 shows a fragment of the system configuration metamodel. A system configuration is the composition of artifacts necessary to make computer systems perform their intended functions [DH07].

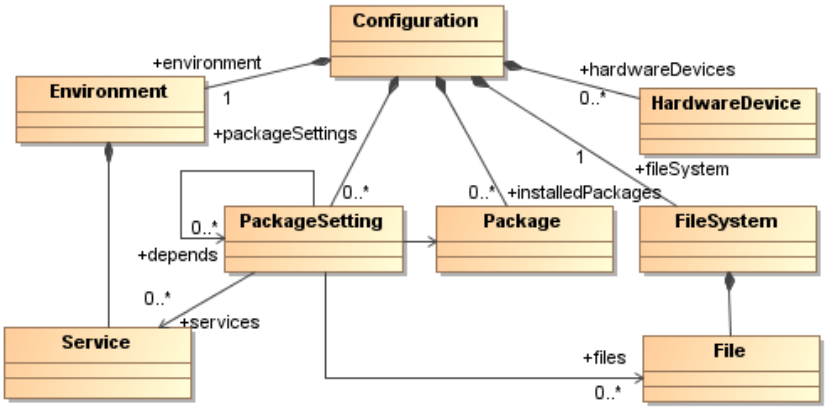

Figure 2.3: Graphical representation of the Configuration metamodel

2.4 Model Injection

The model-based approach to simulate FOSS system updates outlined in the previous section relies on the existence of both a system model which represents an abstraction of the configuration being considered, and the models of the packages which have to be installed. The applicability of the approach depends on the availability of such models which have to be generated necessarily in automatic way. In this respect specific tools, typically named model injectors, which generate models starting from existing artifacts have to be devised. Figure 2.4 shows the role of the injectors in the Mancoosi model-driven approach. The figure highlights the need for two different injectors: one for injecting the configuration of the running system and the other for injecting the packages to be installed and/or upgraded. The injector of the system configuration has the role of extracting a model from the system that is running. The other injector, namely the packages injector, is used by the simulator since it retrieves the information that are needed for simulating the upgrade on the previously injected and synchronized system's model.

In general, model injection can be seen as a solution to the need for modernizing legacy systems in order to keep them up-to-date. The problem is that the legacy systems typically resist

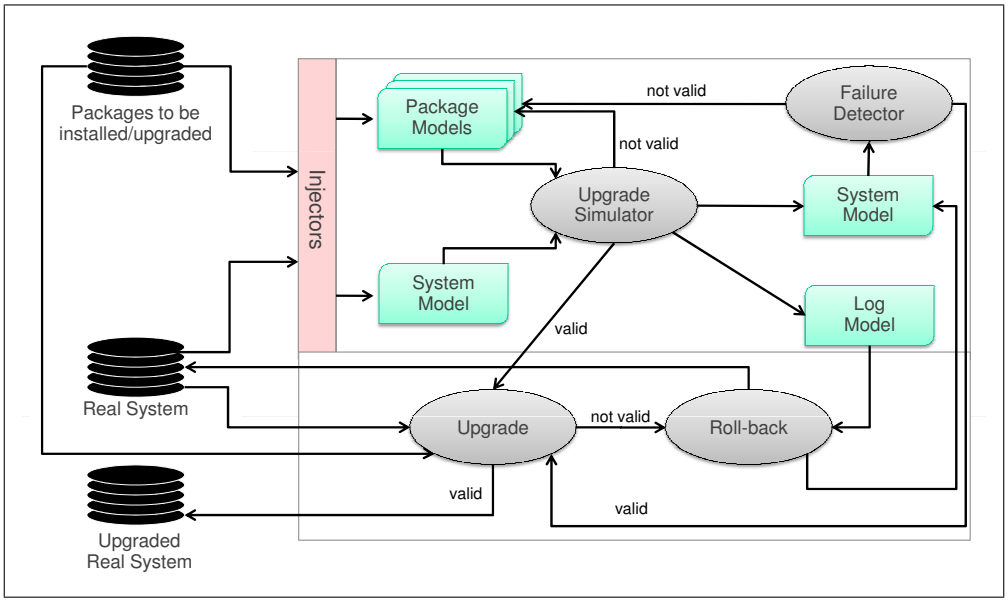

Figure 2.4: The role of the injectors in the Mancoosi model-driven approach

evolution because the capability to adapt them has diminished through factors not exclusively related to their functionality. Reasons for this can be: (i) the difficulty to understand and cost-effectively maintain such a system, (ii) its design models which follow outdated standards, (iii) its ability to interoperate with other systems, or (iv) its dependence on undesired technologies or architectures. There is therefore a need to be able to understand and reverse engineer software assets for the purpose of extracting information and models needed for effective maintenance. Unfortunately, in general modernization is not a cheap task. Moreover, the dimension of the system often impedes handmade modernization. As a consequence, several modernization projects are not finished in a planned time and budget or in extremely cases they are abandoned [sPL03]. This calls for automated modernization. Related to the modernization problem there is the Reverse Engineering (RE) [Eil05] problem that only aims at discovering the technological principles of a system through analysis of its structure and its functioning. In other words, modernization in some sense involves both reverse and forward engineering since it aims at extracting a model and then producing a new modernized model.

Over the last years, several approaches for extracting models from software artifacts have been proposed even though the optimal solution which can be used for any situation does not exist yet [JJJ08]. The complexity of the problem relies on the limitation of current lexical tools which do not provide the proper abstractions and constructs to query code and generate models with respect to given metamodels. Some approaches like [WK06, Eff06] focus on generating metamodels from grammars but they have some drawbacks that may restrict its usefulness, such as the poor quality of the automatically generated metamodel [JJJ08]. Approaches like [JBK06] enable the automatic generation of injectors starting from annotated metamodels with syntactic properties. However, they do not permit reuse of existing grammars written for well-known parser generators. Techniques like [JJJ08] propose specific languages to query software artifacts and generate models according to specified source-to-model transformation rules. Other approaches are:

- MoDisco [Ecl] which defines an infrastructure for supporting model-driven reverse engineering by relying on the concept of *discoverer* which is a piece of software in charge of

analyzing part of an existing system and extracting a model using the MoDisco's infrastructure;

- The work in [FBB+07] proposes a model-driven modernization method developed by Sodifrance IT Modernization company. This method is based on four phases integrated into a tool suite developed by Sodifrance, called Model-In-Action (MIA). These phases are:

  1. derivation of a model, which conforms to a metamodel, of the system from legacy code;
  2. the second phase consists of removing from the model domain specific aspects so to obtain an abstract platform independent model;
  3. the platform independent model conforms to a metamodel called ANT and in the third step the ANT model is transformed into a model specific for the target platform;
  4. in this step the target code is generated from the platform- specific model by means of template-based code generation tools.

- Architecture-Driven Modernization (ADM)[3] [KU07] is an Object Management Group's (OMG's) initiative, which aims to standardize modernization tools and metamodels. OMG characterizes the modernization into three different scenarios [KU07]:

  1. business architecture: this scenario involves the most complicated and comprehensive cases. These scenarios involve models on all levels of abstractions: business, application and data architecture, and technical architecture models;
  2. application/data architecture: in this scenario we deal with changes that impact both the technical and the application-data layer. A typical example is the migration between different platforms. This forces to reorganize the modernized application architecture;
  3. technical architecture: in this scenario we assist to a migration that involves only the technical architecture. A typical example could be the migration of software system from one platform to another or transformation between two languages. It is important to note that in this scenario enhancements cannot impact on system-level or data design;

  Independently from the considered scenario, in [KU07] modernization consists of three main phases:

  1. extracting knowledge from the existing system;
  2. defining the target architecture in order to define a transformation approach;
  3. transformations from existing into a target system.

- Finally, the work in [RGvD06] proposes a model-driven method for managing software migration. The first step of this approach is the extraction of the syntax tree from the original application source code. The next step is the transformation of the model into a pivot language that, in turn, can be mapped into UML.

In the rest of the document we propose an ad-hoc solution for injecting system configurations and packages and to generate the models which are required for simulating system upgrades. In

[3] ADM website: http://adm.omg.org

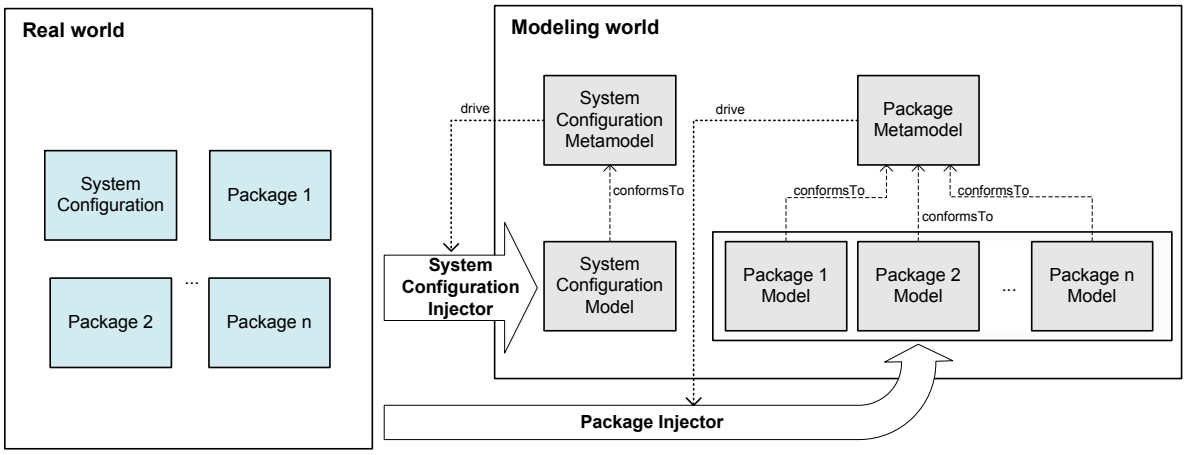

Figure 2.5: Model injection

particular, the proposed technique consists of two different injectors as depicted in Figure 2.5: the *System Configuration injector* is able to generate a model which represents an abstraction of the considered system and which conforms to the *System Configuration Metamodel*. The *Package injector* starting from the set of packages which have to be installed, generates corresponding models which conform to the *Package Metamodel*. Interestingly, the generated package models contain also the maintainer scripts automatically traduced as set of statements of the DSL presented in the Deliverable D3.2.

The approach that we follow in Mancoosi can be categorized in the step-wise approach proposed in [KU07]. The difference is that the target system is the existing system in which we perform an upgrade. Then, we use the first step of the classical modernization approach in order to obtain a model of the system. Once obtained the model we transform it into a target architecture that is the upgraded original system, i.e. the original system goes from one configuration to a new one.

In some sense we use reverse engineering techniques to extract models from the running system. The model can be used for performing some static analysis. Then we can simulate the upgrade on the system and, if the simulation found an error then we discovered that the upgrade cannot be performed due to problems of the models or because the system's configuration does not support the selected upgrade. If the simulation ends positively, then the real upgrade can be performed. In this scenario we are assuming that the system and the model coexist in time and are kept synchronized. In literature this application area is referred as "models at runtime" [BBF09]. This expression puts in evidence that the system can has access at runtime to various models representing the system [JBB09]. As a consequence, the models need to be kept in synchronization with the current state of the system. This is exactly what we do in Mancoosi. The techniques that we present in this remaining of this deliverable are used to extract the model from the system, but, if a model of system already exists then it is possible to use the same technique in order to synchronize the model with the running system. This is extremely important since, even thought we can imagine to instrument each action that causes a system upgrade with suitable operations act to keep the model synchronized, in practical cases this does not work. In fact, numerous are the ways a system can change something that is relevant

for the model, here included the user actions. This testifies the need for having synchronization mechanisms. As explained in the remaining of the deliverable, the synchronization analyzes the system and when discovers a difference with the information stored in the model, it accordingly updates the model. In this way the synchronization is very efficient as shown in Chapter 4.

Chapter 3

System configuration injection

This chapter describes the application which has been conceived for injecting the configuration of running FOSS systems. The approach has been designed to be easily extended and to support any Linux distribution (e.g. Debian, Mandriva, Caixa Magica, etc.). In particular, the implemented system consists of three layers as shown in Figure 3.1: the upper layer consists of injectors which are specific to the considered distribution since, to query the considered system, they use the proper package manager like *dpkg* and *rpm*. The creation and the population of models according to the gathered information is performed by extending and using the elements provided by the *Mancoosi model injection infrastructure* which is independent from the considered distribution and then it is wrote once and for all. Such layer uses the *Mancoosi model management* one which provides the means for creating and managing models which conform to the Mancoosi metamodel summarized in the previous chapter. Moreover, we engineered the overall injection architecture thus supporting also "horizontal" extensions. More precisely, the *Mancoosi Model Injection Infrastructure* can be extended by adding more classes by enlarging the type of runtime errors that can be identified.

The remaining of the chapter is organized as follows: Section 3.1 introduces the Eclipse Modeling Framework (EMF)[1] which underpins the overall injection architecture. Section 3.2 presents in detail the layer which is devoted to the management of Mancoosi models, whereas the model injection infrastructure is described in Section 3.3. The prescriptions which have to be considered to implement distribution specific injectors are described in Section 3.4.

[1]EMF project Web site:http://www.eclipse.org/modeling/emf/

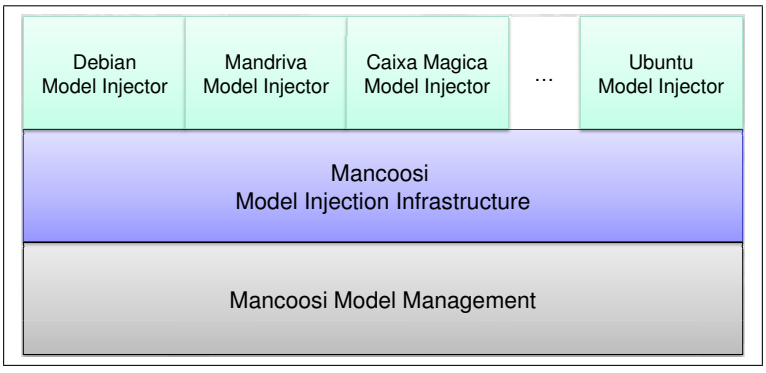

Figure 3.1: The Mancoosi model injection architecture

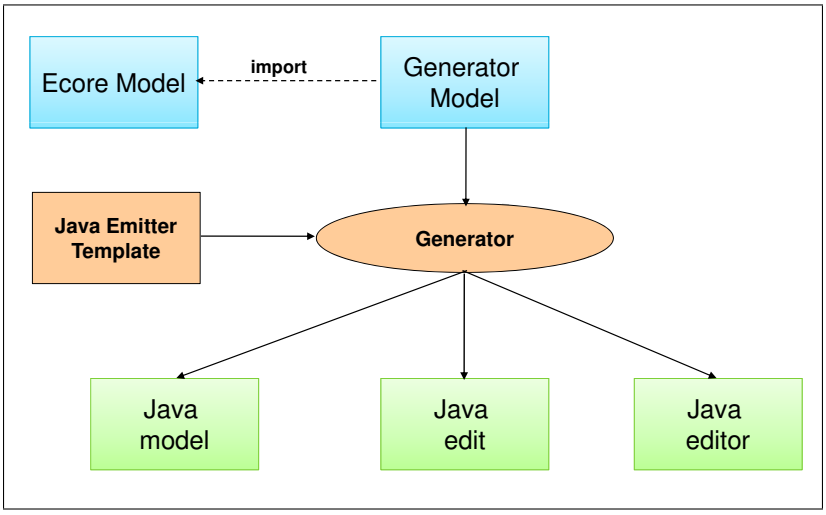

Figure 3.2: The Eclipse Modeling Framework Toolkit

3.1 Eclipse Modeling Framework

This section presents an overview of the technologies used to implement the overall Mancoosi model injection. The approach is implemented by making use of the Eclipse development platform and of some of its plugins, as explained in the following. In particular, Eclipse[2] is an open source development platform comprised of extensible frameworks and tools for building, deploying and managing software across the lifecycle. The Eclipse open source community has over 60 open source projects. One of these projects is the Eclipse Modeling Framework (EMF) that is a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF brings together and integrates modeling and programming without forcing a separation between high-level modeling and low-level implementation programming. Therefore, within EMF, modeling and programming can be considered the same thing. On one side EMF provides a solid, high-level way both to communicate the design and to generate part (if not all) of the implementation code, on the other side a programmer can still maintain its programming attitude. Then, the aim of EMF is to mix modeling with programming in order to maximize the effectiveness of both.

All the main building blocks of the EMF Toolkit are depicted in Figure 3.2. A key role is played by the Ecore model which represents the metamodel which has been developed by analyzing the considered applicative domain and capturing the relevant concepts in terms of meta-classes and relations among them. Such a model conforms to the Ecore metamodel which is reported in Figure 3.3. The Ecore metamodel can be considered the Eclipse implementation of the OMG Meta Object Facility (MOF) [Obj03]; a fragment is reported in Figure 3.3 and the most important elements are described in the following [BSM+03]:

– *EClass:* this is a central element in the overall metamodel since it permits the specification of classes, which are the nodes of an object graph. Classes are identified by name and can contain a number of features (i.e. attributes and references). They can refer to a number of other classes as its super-types, and can be abstract; in this case an instance cannot be created;

[2]Eclipse project Web site: `http://www.eclipse.org`.

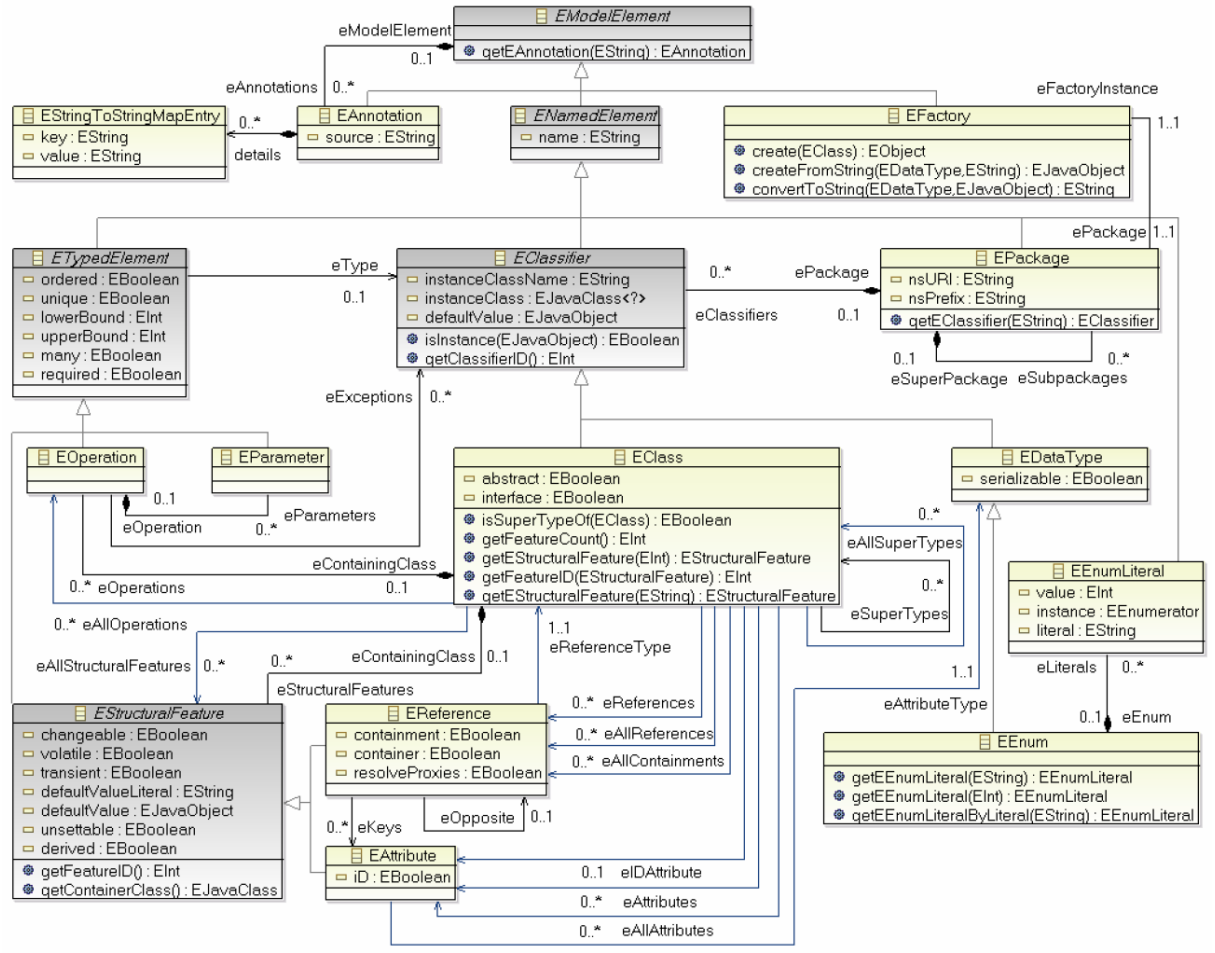

Figure 3.3: Ecore metamodel

– *EAttribute:* this element models attributes, which are the leaf components of an object's data. They are identified by name and they have a type;

– *EReference:* this element models one end of an association between two classes. They are identified by name and type, where that type represents the class at the other end of the association. Bidirectionality is supported by pairing a reference with its opposite, i.e. a reference in the class representing the other end of the association. Lower and upper bounds are specified in the reference for multiplicity. A reference can support a stronger type of association called containment;

– *EDataType:* this element models simple types whose structure is not modeled. They are identified by name and are most commonly used as attribute types;

– *EStructuralFeature:* it is the common base class for attribute and reference which can be characterized by the following Boolean attributes:

  - *Changed,* whether the value of the feature can be modified;
  - *Derived,* whether the value of the feature has to be computed from those of other related features;
  - *Transient,* whether the value of the feature is omitted from the object's persistent seri-

alization;

- *Unsettable,* whether the value of the feature has an unset state distinguishable from the state of being set to any specific value;
- *Volatile,* whether the feature has no storage field generated in the implementation class.

– *EPackage:* it models packages, containers for classifiers, i.e. classes and data types. A package's name needs not be unique; its namespace URI is used to uniquely identify it. This URI is used in the serialization of instance documents, along with the namespace prefix, to identify the package for the instance;

– *ETypedElement:* it models elements which have a type, e.g. attributes, references, parameters, and operations. All typed elements have an associated multiplicity specified by their `lowerBound` and `upperBound`;

– *EEnum:* it models enumeration types, which specify enumerated sets of literal values;

– *EEnumLiteral:* it models the members of enumeration type's set of literal values. An enumeration literal is identified by name and has an associated integer value as well as literal value used during serialization.

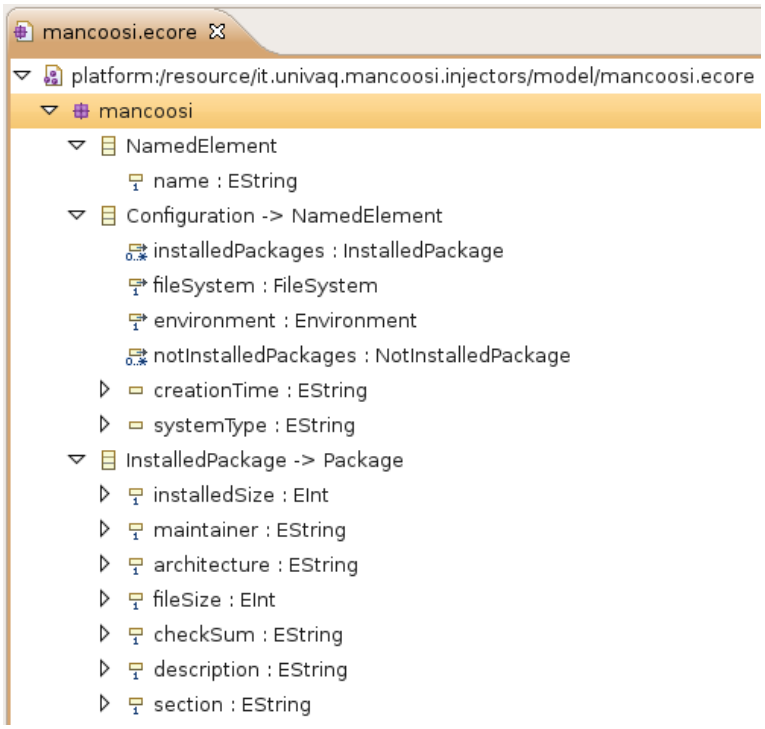

Figure 3.4: Fragment of the Mancoosi metamodel developed in Ecore

Such elements are used to conceive metamodels which play a key role in MDE, since they are the formal definition of well-formed models, or in other words they constitute the languages by which a given reality can be described in some abstract sense [Bó5]. A small fragment of the Mancoosi metamodel which conforms to the Ecore metamodel outlined above is reported in Figure 3.4. In particular, the figure depicts the metaclass *NamedElement* which is a supertype of the metaclasses *Configuration*, and *InstalledPackages*. For each metaclass, a number of structural features are specified like the attribute *creationTime* of the metaclass *Configuration*

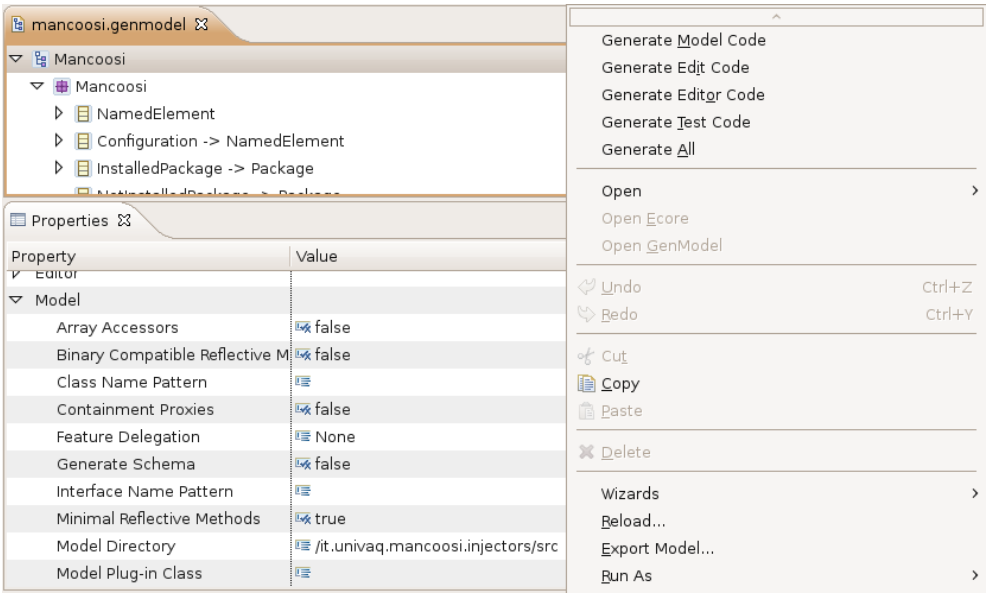

Figure 3.5: Sample Generator Model

which is used to maintain when a given configuration model has been created, or the reference *installedPackages* which will maintain all the installed packages of a given system.

Once a metamodel has been defined, it is used by a *Generator Model* as depicted in Figure 3.2 which specifies some properties of the code to be generated, as for instance, the name of the packages which have to be generated, the path in the filesystem of the considered system which will contain the generated code, etc. For instance, the generator model fragment reported in Figure 3.5 imports the Mancoosi metamodel previously sketched and provides additional information related to the code generation. In the reported example, the source code of the model package, which will be introduced in the rest of the section, has to be generated in the path /it.univaq.mancoosi.injectors/src.

A generator model like the one reported in Figure 3.5 is the input of a code generator which is able to produce the Java implementation of the provided metamodel which consists of the following three main components:

- *model*: it is a Java library which represents the core model manipulation and persistence implementation. In particular, all the facilities to create, modify, validate, serialize, and de-serialize models conforming to the given metamodel are automatically generated and contained in such a component;

- *edit*: it is a Java library which contains methods for querying and manipulating models to be offered to graphical editors which do not have direct access to the model;

- *editor*: it is an automatically generated tree based editor like the one reported in Figure 3.6.a which has been generated from the Mancoosi metamodel. In the editor a sample Mancoosi model is reported and represents an Ubuntu system configuration consisting of many installed packages like alacarte, apt-utils, etc. For each installed package, a number of properties are also reported as for instance the conflicts, dependencies, the configuration files etc.

Concerning the developed injectors for system configurations, the EMF toolkit reported in Figure 3.2 has been adopted to generate the *Mancoosi Model Management* layer depicted in Figure 3.1. In particular, starting from the Mancoosi metamodel, the mancoosi.model component has been generated as explained in the next section.

3.2 The Mancoosi model management

As previously said, the EMF toolkit presented above has been used for the implementation of the overall model injection architecture, especially to generate the Java code which is necessary to manipulate models conforming to the Mancoosi metamodel. In particular, the generated *model* component contains two main Java packages named mancoosi and mancoosi.impl as depicted on the right-hand side of Figure 3.7. Being more precise, the package mancoosi contains Java interfaces defined according to the metaclasses of the Mancoosi metamodel. For example, the metaclass *Configuration* induces the generation of the Java interface Configuration in the package mancoosi. For each structural feature of the source metaclass, corresponding getters and setters are also generated (e.g. the methods setEnvironment(Environment) and getEnvironment() are generated from the meta-relation environment in the meta-class Configuration). The package mancoosi.impl contains the Java classes which implement all the interfaces in the package mancoosi (e.g. the class ConfigurationImpl implements the interface Configuration). Moreover, according to the *Abstract factory pattern* [GHJV95], the MancoosiFactoryImpl class is generated in order to provide a factory class that programmers have to use to create new model elements.

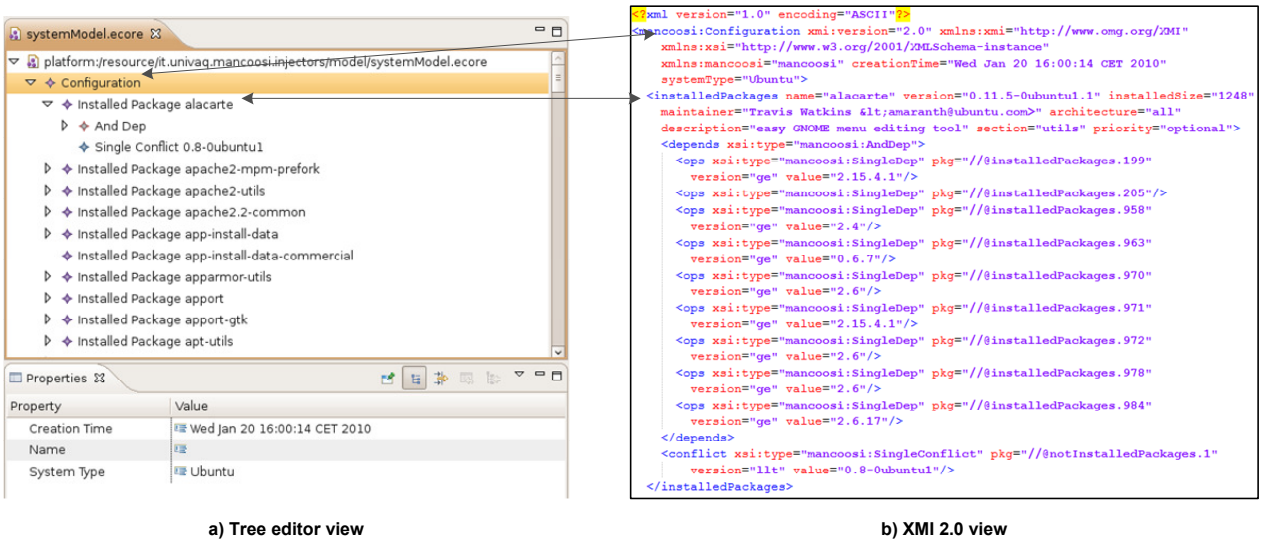

Figure 3.6: Sample Mancoosi model

A sample use of the code generated from the Mancoosi metamodel is reported in Listing 3.1. The sample Java class Injector uses the methods provided by the generated class Configuration to create a new configuration and to set its structural features (see lines 10-13). A new environment is also created and the containment relation between the metaclasses *Configuration* and *Environment* is considered by executing the method setEnvironment (see line 14).

Listing 3.1: Sample use of the Mancoosi model management layer

```
1  import mancoosi.MancoosiFactory; import mancoosi.Configuration;
2  import mancoosi.Environment; ...
3
4  public class Injector {
5
6      public static void main(String[] args) {
7          ...
8          MancoosiFactory factory = MancoosiFactory.eInstance;
9          Configuration configuration = factory.createConfiguration();
10         configuration.setCreationTime((new GregorianCalendar()).getTime().toString());
11         configuration.setSystemType("Ubuntu");
12
13         Environment environment = factory.createEnvironment();
14         configuration.setEnvironment(environment);
15         ...
16         URI fileURI = URI.createFileURI("model/systemModel.ecore");
17         Resource resource = new XMIResourceFactoryImpl().createResource(fileURI);
18         resource.getContents().add(configuration);
19         resource.save();
20     }
21 }
```

The generated code from the Mancoosi metamodel is also able to serialize and de-serialize models in XML documents as for instance the model in Figure 3.6.b which has been obtained by executing a code fragment like the one in lines 16-19 of Listing 3.1 which saves the sample created configuration in the file systemModel.ecore located in the directory ./model.

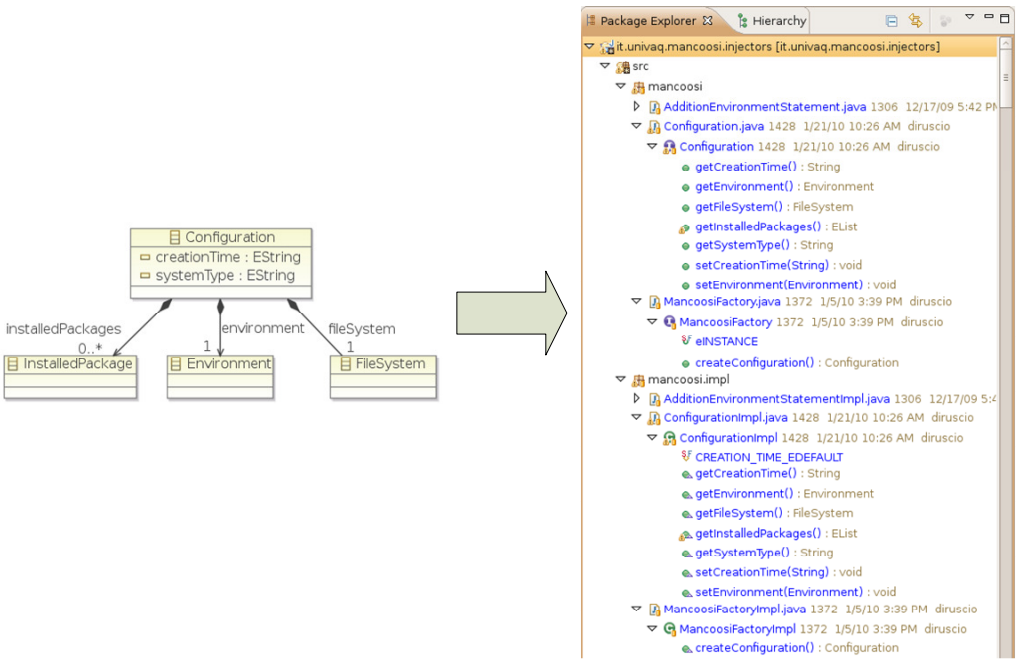

Figure 3.7: Small fragment of the Mancoosi model management layer

3.3 The Mancoosi model injection infrastructure

The Mancoosi model management layer presented in the previous section underpins an injection infrastructure which contains a library of Java abstract classes which embody common

model manipulation operations which are required by any injector independently from the considered distribution. Figure 3.8 depicts the main building blocks of the model injection infrastructure which consists of two main Java packages named mancoosi.injectors.managers, and mancoosi.injectors.utils. The former contains a number of classes each devoted to the management of a specific aspect of a Linux distribution, like the file system, packages, alternatives, etc. The latter contains auxiliary classes which are used internally by the managers as will be explained in the following.

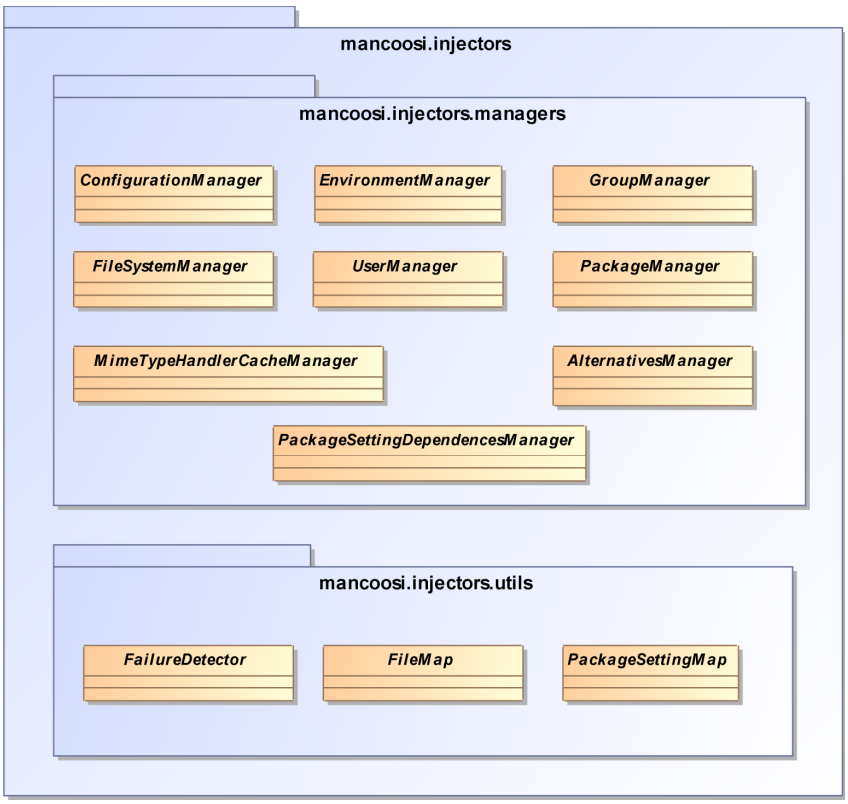

Figure 3.8: Fragment of the Mancoosi model injection infrastructure

Hereafter, all the classes reported in Figure 3.8 are explained in depth by means of some explanatory examples. It is worth noting that all the managers implement the singleton design pattern [GHJV95] in order to guarantee that only one instance for each manager exists in the running injector.

Configuration Manager It is used to manage the abstract representation of the system configuration being injected. In particular it provides the methods to get and set the single instance of the metaclass *Configuration* which is populated by the other managers during the overall injection phase (see Figure 3.9). The method getInstance is contained also in all the other managers and it is used to get the single available instance of the manager. The method synchronize plays an important role since it can be executed to synchronize the configuration represented in the model with the real one. In particular, the execution of the method synchronize will invoke the same method of all the other managers to update models in order to reflect the modifications which have been performed on the real system. For instance, if the user

accidentally removes a configuration file or some executables which are important for the correct use of the system, by means of the synchronization phase the user is informed about missing files or fallacious situations and the models are consistently updated. Vice versa, if the model has been manually modified, the invocation of the method synchronize will incrementally update it by recovering its consistency with the current system configuration. It is important to note that we do not allow changes made by hand on the models; moreover in case of conflicts the models will be updated to reflect the system configuration.

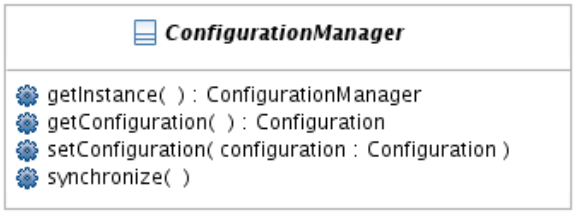

Figure 3.9: Configuration Manager

Environment Manager Its structure is like that of the configuration manager previously described (see Figure 3.10). The main purpose of this manager is maintaining a single instance of the metaclass *Environment* and having it available everywhere in the injector being implemented. The environment element is used to represent shared libraries and running processes of the considered system.

Figure 3.10: EnvironmentManager Manager

Alternatives Manager The concept alternative refers to 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 categorizes packages that provide similar functionality. 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, then the operating system and user can select which email client they would like to use. Alternatives can be represented in a Mancoosi model by means of instances of the metaclass *Alternative* and the *Alternatives Manager* class is introduced to create and maintain them (see Figure 3.11). The abstract method createAlternativesFromSystem plays a key role since it creates all the modeling elements which are required to represent the alternatives installed in the considered system. The method is abstract since it is distribution

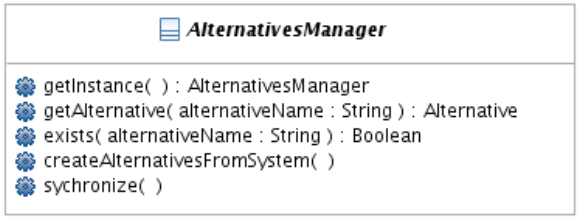

Figure 3.11: AlternativesManager Manager

dependent and each injector has to implement it. For instance, in Debian based distributions, the installed alternatives can be retrieved by looking at the file /etc/alternatives.

FileSystem Manager It provides common methods which are typical in the management of modeling elements representing the file system. All the methods of the abstract class FileSystemManager have an implementation which eventually can be customized by properly extending the class as it will be described in the next section. In particular, apart from the operations related to the singleton pattern and to the synchronization facility, the class provides the following methods (see Figure 3.12):

- createFile(path : String) : File, given a string representing a path in the file system, corresponding instances of the metaclass *File* are created as in the example in Figure 3.13;
- deleteFile(path : String) : Boolean, it removes the instances of the metaclass *File* corresponding to the files located in the path provided as parameter. For instance, the execution of the code deleteFile(/etc/acpi/events/ac) will remove all the file element ac contained in the modeling element events;
- getFile(path : String) : File, this method is able to retrieve from the model the element which corresponds to the path provided as parameter;
- exists(path : String) : Boolean, it checks whether exists a modeling element corresponding to the file located in the path given as parameter.

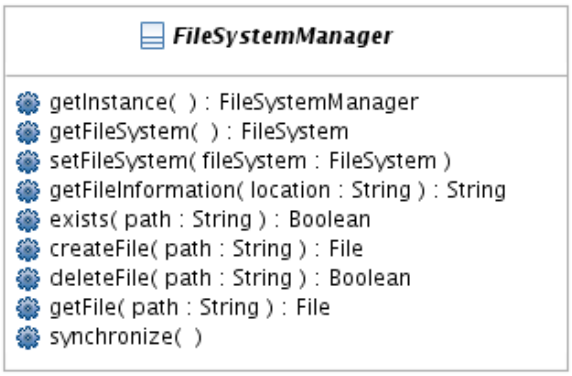

Figure 3.12: FileSystem Manager

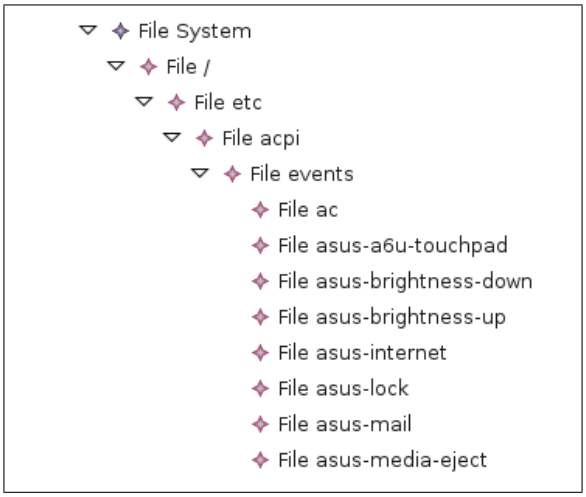

Figure 3.13: Sample file system model

The FileSystem Manager supports also symbolic links as dangling symbolic links would be easy spot for runtime failures.

As previously said, a specific injector will extend FileSystemManagers abstract class and eventually introduce new methods for file system manipulations specific to the considered distribution.

Group Manager This class provides the operations which are typically required for managing user groups of an installed system (see Figure 3.14). In particular, apart from the methods of the singleton design pattern and that provided for synchronizing the modeled groups with those really existing in the considered system, the group manager provides also the following methods:

- addGroup(groupName : String) : Group, given a string which represents the name of a group, a new instance of the metaclass *Group* is created;
- getGroup(groupName : String) : Group, this methods is able to retrieve the modeling element corresponding to the group identified by the string given as parameter;
- getAllGroups() : Group[], it retrieves all the groups represented in the considered configuration model;

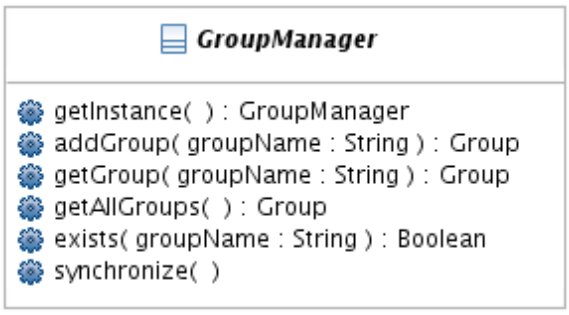

Figure 3.14: Group Manager

User Manager The management of system users is very similar to that of the groups previously presented (see Figure 3.15). The main difference between these two classes is on the CreateUsersFromSystem() method which injects all the users of the real system and updates the models by adding corresponding modeling elements. This method makes use of the group manager in order to add groups in the model according to the information of each user. A model fragment which represents some of the users and groups of a real system is reported in Figure 3.16.

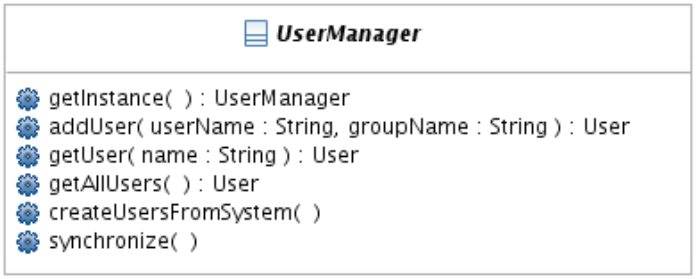

Figure 3.15: User Manager

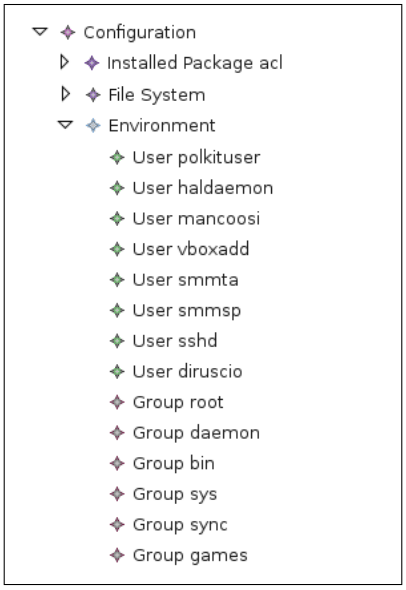

Figure 3.16: Sample model with injected users and groups

Package Manager This manager plays a key role in the overall infrastructure since the configuration injection starts by considering first the installed packages. In this respect, the method createPackageElementsFromSystem() in Figure 3.17 updates the model being created by adding all the information related to the installed packages. This is an abstract method since the way the installed packages have to be gathered from the system heavily depends on the considered distribution. For instance, if we are considering a Debian-based system, the package manager dpkg can be used to query the configuration and to retrieve information about the installed packages. In case of RPM-based distribution, the package manager rpm can be considered. The

other abstract methods of the package manager which have to be implemented with respect to the the kind of considered system are the following:

- setFeaturesOfInstalledPackages(), this abstract method has to be implemented by each new injector to set all the structural features of the installed packages consisting of the name, architecture, configuration files, package dependencies, etc. Such method can use the other abstract methods which are described in the following;
- processPackageMetadataLine(String aLine, InstalledPackage pkg), some of the features of the installed packages can be defined by means of this abstract method which has to be defined by each injector in order to process all the metadata of the installed packages. Since the ways for retrieving the package information depend on the considered distribution, this method is provided as abstract. Then each injector will implement it with respect to the package format of the considered distribution. Such a method can use the abstract methods which are presented hereafter;
- processConfFiles(List<String> conffilesBlock, InstalledPackage installedPackage), for each installed package, all the corresponding configuration files have to be injected in the configuration model. In this respect, this abstract method is provided and each injector will implement it by using the proper tools which the considered distribution offers to query the system and retrieve the information related to the configuration files of each package;
- createSingleDeps(String[] singleDeps, Dependence owner), the dependencies of each retrieved package can be the conjunctions and/or disjunctions of strong atomic dependencies. This abstract method has to be implemented in order to retrieve the single dependencies of each package by considering its metadata;
- createSingleConflicts(String[] singleConflicts, Conflict owner), the installed packages can be in conflict with other packages. As for the dependencies, the conflict specification can involve single conflicts which are in conjunctions and/or disjunctions. This abstract method has to be implemented by each injector in order to retrieve the single conflicts by looking at the metadata of the considered packages;

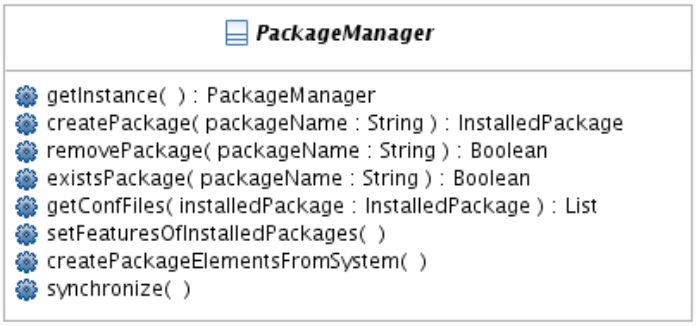

Figure 3.17: Package Manager

According to the *Package* metaclass of the Mancoosi metamodel, all the configuration files of a given package are contained in an instance of the *PackageSetting* metaclass. The management of dependencies among package settings which are not managed by common package managers are considered by *PackageSettingDependenciesManager* described in the following.

PackageSettingDependencies Manager It queries the given configuration and gathers possible dependencies between configuration files; these files may give place to inconsistent states if not properly managed. In particular, even though the dependencies between packages are not defined, there are some configuration files which depend on other configuration files belonging to different packages. For instance, the Apache Web server does not depend on PHP5 module (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. Unfortunately, the Apache package does not depend on PHP5 even though their configuration files might have dependencies. Current package managers do not support the dependencies between configuration files, hence the dependencies which can occur on a given configuration are not directly available but they have to be retrieved.

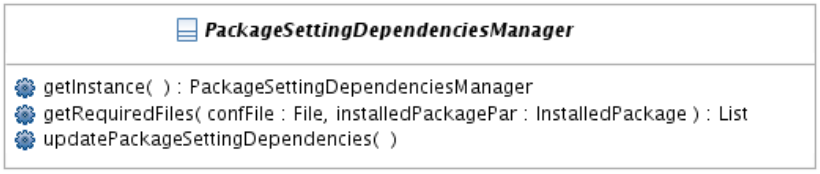

Figure 3.18: Package setting dependencies Manager

The abstract class *PackageSettingDependenciesManager* shown in Figure 3.18 provides the programmer with some methods which have to be implemented with respect to the specific system being injected. The main methods provided by the manager are the following:

- getRequiredFiles(File confFile, InstalledPackage installedPackagePar) : List <File>, this abstract method returns all the configuration files which are used in the one passed as parameter;
- updatePackageSettingDependencies(), this method updates the dependencies between the package settings already created in the configuration model by exploiting the method getRequiredFiles(). Being more precise, for each configuration file of each installed package, it checks whether such a file depends on others in the system.

A concrete application of such a manager will be described in the next chapter.

MimeTypeHandlerCache Manager In the Mancoosi metamodel the metaclasses *MimeTypeHandlerCache* and *MimeTypeHandler* are provided to maintain the cache of the mime type handlers installed in the considered system. This manager provides the methods to create and have access to the modeling element which have the references to all the mime type handlers installed in the system and represented in the model. In particular, the method createMimeTypeHandlerCacheFromSystem() queries the considered system, gathers all the information related to the mime types and, finally, injects them into the model. Such a method is abstract since each distribution maintains the information related to the mime types in different ways. Thus, each new injector will implement it according to the prescriptions of the considered distribution.

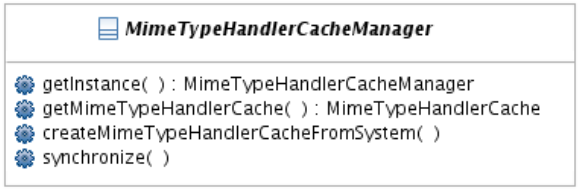

Figure 3.19: Mime type handler Cache Manager

The set of managers that has been provided can be extended as introduced at the beginning of this chapter by adding the support for new entities, such as XML catalog entries, the registry of available Python modules, crontab entries, documentation registries, menu entries, etc. It is important to note that the larger the set of classes that will be added, the larger the class of runtime errors that can be identified.

The `mancoosi.injectors.utils` package As depicted in Figure 3.8 the managers described above are contained in the Java package `mancoosi.injectors.manager`. To increase the performance of the overall injection approach some utility classes have been implemented (and located in the package `mancoosi.injectors.utils`) especially to reduce the time required to retrieve the elements from the model being considered. For instance, the classes `FileMap` and `PackageSettingMap` implement two data structures which have been conceived to maintain file and package setting model elements, respectively, and to reduce the time required to retrieve them especially when big configuration models have to be manipulated. The package `mancoosi.injectors.utils` contains also the class `FailureDetector` which provides the user with operations able to perform static analysis on given configuration models and to identify possibly system inconsistencies which may give place to upgrade faults or malfunctions. Some of the available failure detection possibilities which are currently supported are explained in depth in Chapter 6.

Manager	Required customizations
Configuration	None
Environment	None
FileSystem	None
Group	None
MimeTypeHandlerCache	- `createMimeTypeHandlerCacheFromSystem`
Package	- `createSingleDeps` - `createSingleConflicts` - `processPackageMetadataLine` - `processConfFiles` - `setFeaturesOfInstalledPackages` - `createPackageElementsFromSystem`
PackageSettingDependencies	None
User	None

Table 3.1: Required customizations to implement model injectors

3.4 Developing distribution-dependent model injectors

The Mancoosi model injection architecture reported in Figure 3.1 has been conceived to support any distribution by identifying common elements and manipulations in an intermediate layer and provides the means to use and extend it with respect to the specificities of the considered system. In particular, implementing a new injector consists of extending all the managers described in the previous section and providing the implementation of all the available abstract methods. Table 3.1 reports all the elements which have to be implemented to support new distributions. Such a table will be considered in the next chapter which describes the implementation of a developed injector to support Ubuntu systems.

Chapter 4

Configuration injectors for Debian-based systems

In this chapter we describe the injector which has been implemented to support the injection of Ubuntu 9.10 systems. The complete implementation of the injector is available online[1]; in this chapter only the essentials will be discussed with the aim to provide guidelines for developing new injectors for other Linux distributions.

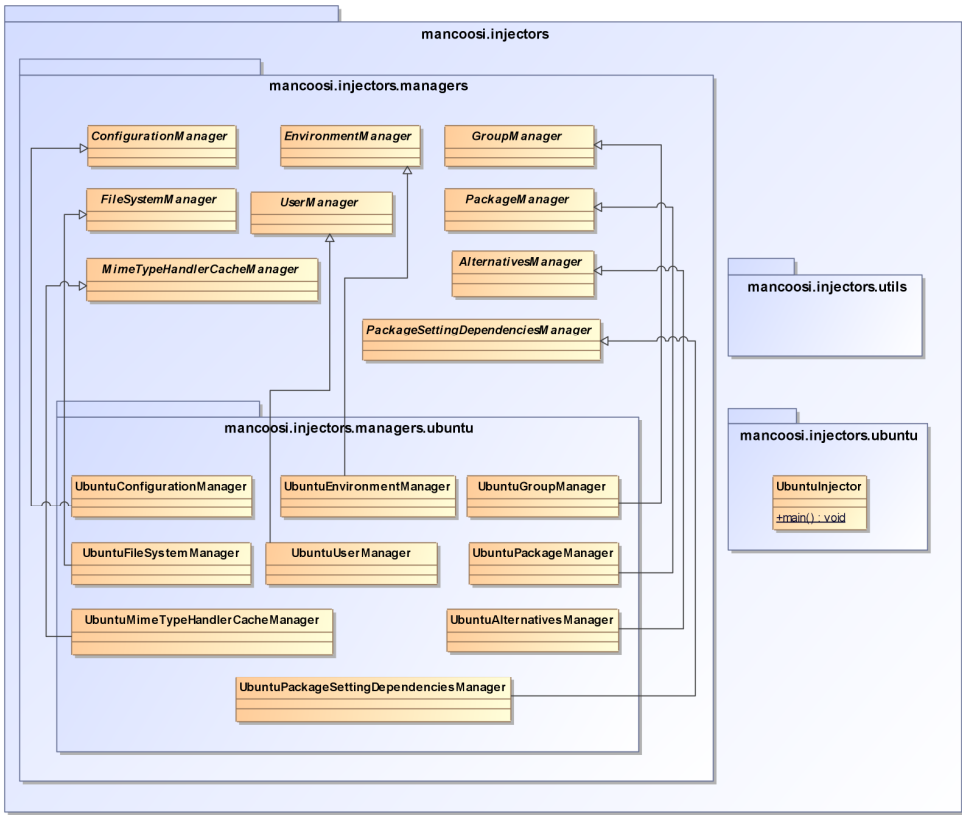

Figure 4.1: Fragment of the Ubuntu model injector architecture

[1] http://www.mancoosi.org/reports/d2.2-eclipseModelInjection.tar.gz

As said in the previous chapter, to implement an injector for supporting a new Linux distribution, the injection infrastructure presented in the previous chapter has to be considered by extending the provided classes and implementing their abstract methods. According to such prescriptions, the Ubuntu 9.10 injector has been implemented as depicted in Figure 4.1. Listing 4.1 reports a fragment of the main method of the class UbuntuInjector which uses all the implemented Ubuntu managers to inject a running systems in a corresponding configuration model.

Listing 4.1: Fragment of the main method of the UbuntuInjector class

```
1  // Model Initialization
2  Configuration configuration =
3  UbuntuConfigurationManager.getInstance().getConfiguration();
4
5  Environment environment =
6  UbuntuEnvironmentManager.getInstance().getEnvironment(); FileSystem
7  fileSystem = UbuntuFileSystemManager.getInstance().getFileSystem();
8  MimeTypeHandlerCache mimeTypeHandlerCache =
9  UbuntuMimeTypeHandlerCacheManager.getInstance().getMimeTypeHandlerCache();
10 environment.setMimeTypeHandlerCache(mimeTypeHandlerCache);
11 configuration.setEnvironment(environment);
12 configuration.setFileSystem(fileSystem);
13
14 // System users are added in the model
15 UbuntuUserManager userManager = (UbuntuUserManager)
16 UbuntuUserManager.getInstance(); userManager.createUsersFromSystem();
17
18 // All the installed packages are retrieved and corresponding modeling elements are
       ↪created
19 PackageManager pkgManager = UbuntuPackageManager.getInstance();
20 pkgManager.createPackageElementsFromSystem();
21
22 // Dependencies among package settings are retrieved
23 UbuntuPackageSettingDependenciesManager.getInstance().updatePackageSettingDependencies()
       ↪;
24
25 // Mime types and their handlers are represented in the model
26 UbuntuMimeTypeHandlerCacheManager.getInstance().createMimeTypeHandlerCacheFromSystem();
27
28 // All the alternatives installed in the system are represented in the model
29 UbuntuAlternativesManager.getInstance().createAlternativesFromSystem();
30
31 configuration.setCreationTime((new
32 GregorianCalendar()).getTime().toString());
33 configuration.setSystemType("Ubuntu"); ...
```

According to the Mancoosi metamodel, a system model has a configuration element as root, and all the other model elements are contained in it such as the environment and the filesystem. In this respect, a new configuration element is created by using the getConfiguration method of the developed UbuntuConfigurationManager (see line 3 in Listing 4.1), then the modeling elements which are used to represent the file system and the environment are created by using the UbuntuFileSystemManager and UbuntuEnvironmentManager classes, respectively. The containment relation between such elements and the previously created configuration is set by using the methods setEnvironment and setFileSystem provided by the Mancoosi model management layer (see lines 11-12). In the same way, the modeling element which will maintain all the mime type handlers installed in the considered system is created by means of the proper manager (see line 8) and specified as contained in the environment element previously created (see line 10). After such an initialization phase, the model is updated by means of data gathered from the real system by using methods which are Ubuntu specific. For instance, the method createPackageElementsFromSystem of the UbuntuPackageManager class retrieves all the installed packages by executing the shell command dpkg -l and properly parses its outcome. For each

package, the corresponding configuration files are retrieved by using the command `dpkg -s`.

The provided managers are able to retrieve also information which is not directly available and needs to be identified by means of a complex navigation on the real system. For instance, the identification of the dependencies between the configuration files is an example and it is performed by the method `updatePackageSettingDependencies` of the `UbuntuPackageSettingDependenciesManager` class (see line 23).

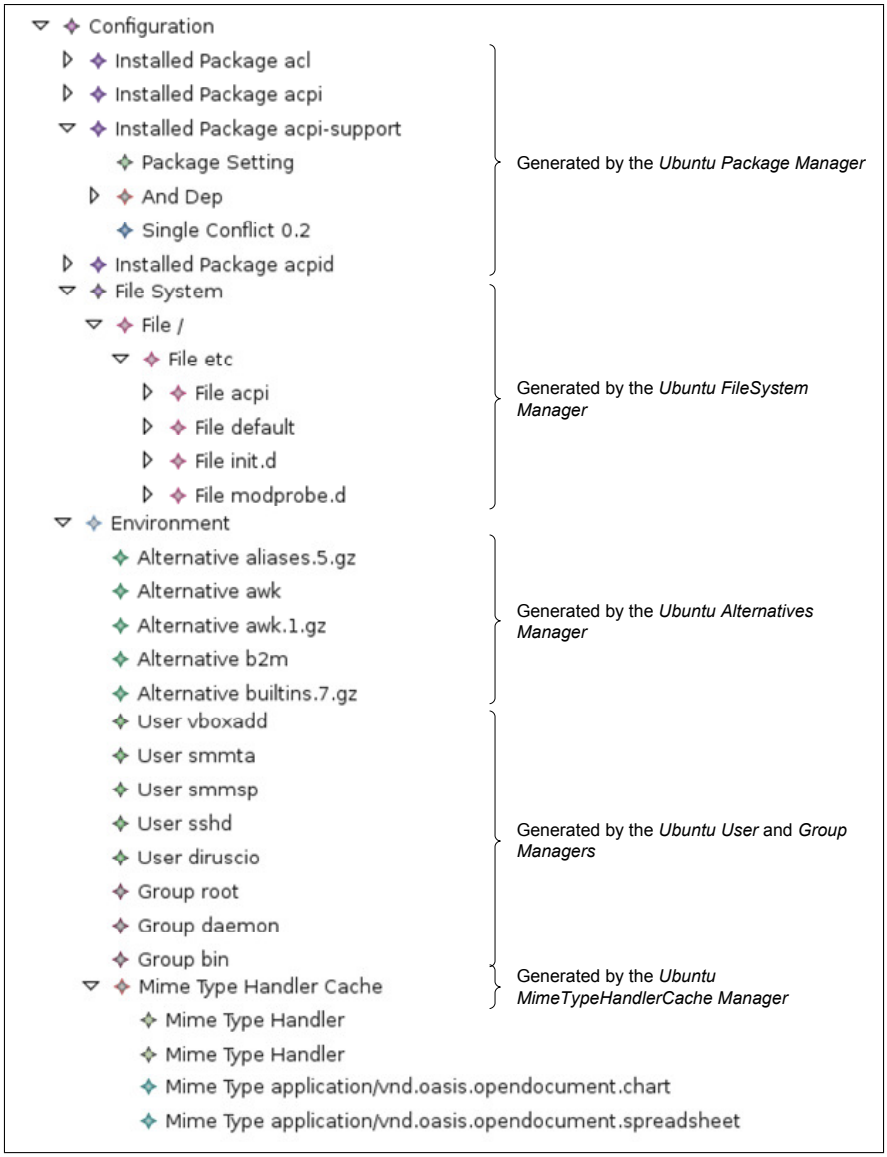

Figure 4.2: Fragment of an injected Ubuntu configuration

Figure 4.2 reports a small fragment of a configuration model generated by executing the Ubuntu 9.10 injector on a running system consisting of ≈1.300 installed packages. The model is reported by using a tree-based editor which is part of the Mancoosi model management layer. However, since the generated Mancoosi models will be manipulated and analyzed by means of further provided tools (as described in Chapter 6), supporting their visualization by means of specific editors is not relevant even though technically possible by using proper technologies like GMF[2],

[2]Eclipse Graphical Modeling Framework (GMF) project: http://www.eclipse.org/gmf

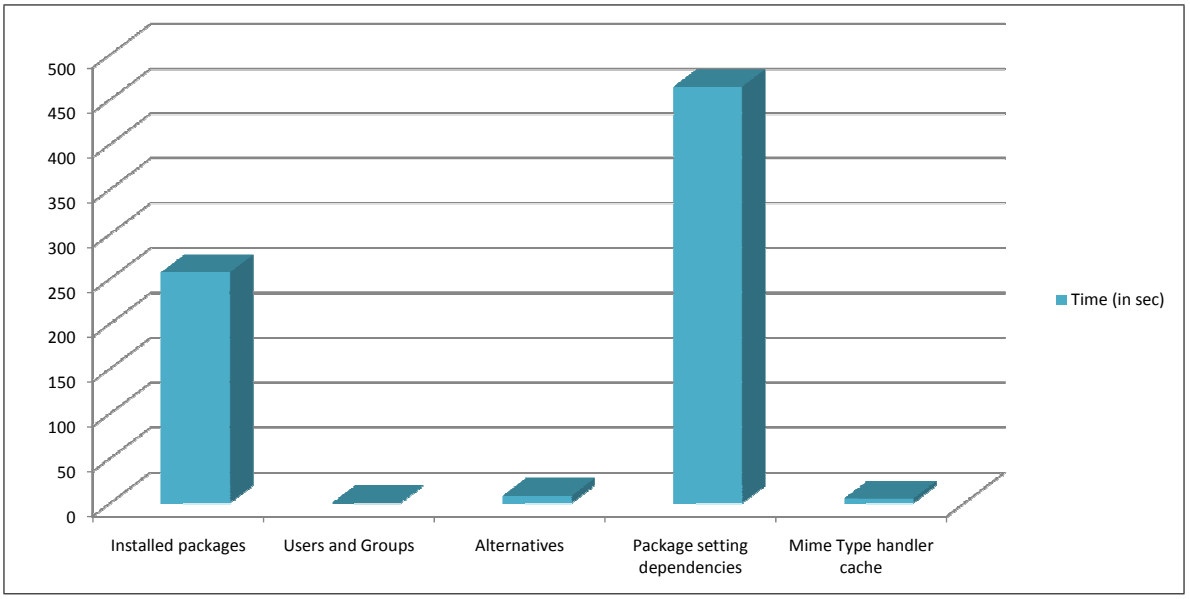

Figure 4.3: Time required to inject a running Ubuntu 9.10 system

TCS[3], etc. which permit to specify ad-hoc concrete syntaxes for given metamodels. For instance, by using such technologies it is possible to implement tools to query Mancoosi models and show to the users only some views of them like the installed packages and their inter-dependencies, the services which are executed when the system is started, etc.

The generation of a Mancoosi configuration model may take some time depending on the hardware configuration of the considered system. Figure 4.3 shows the time which has been required to inject an Ubuntu 9.10 system, running on a virtual machine with 512MB of RAM upon Virtual Box[4]. As shown in Figure 4.3 the injection of directly available elements is really fast as for instance the users and the groups of a system, the configured alternatives, or the installed mime type handlers. More time is required to inject the installed packages especially because all their

[3]Textual Concrete Syntax (TCS) progect: http://www.eclipse.org/gmt/tcs/
[4]VirtualBox Website: http://www.virtualbox.org/

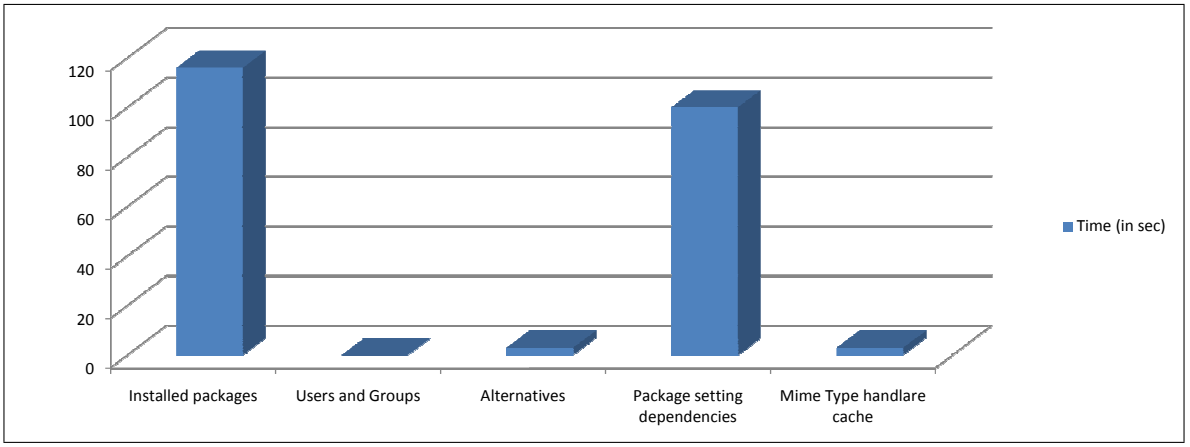

Figure 4.4: Time required to inject an Ubuntu 9.10 system running on a faster machine

configuration files have to be retrieved. The operations which take longer are related to the injection of information which has to be retrieved by means of complex system navigations as for instance the dependencies between the configuration files (see *Package setting dependencies* in Figure 4.3).

We have executed the model injection approach also on an Ubuntu 9.10 system running on a machine with a quad-core processor, 2.83Ghz, 4GB of RAM with ≈1.400 installed packages. Figure 4.4 reports the execution time of such an injection which has been completed in less than 120 seconds.

As said in the previous chapter, the overall injection infrastructure has been designed to support also the synchronization between real systems and corresponding configuration models. In fact, each manager provides the user with the method `synchronize` to update the model according to the modifications operated on the system configuration without to force a complete regeneration of the model. The experiments we performed on the machines previously considered took few seconds for performing the synchronization. The rationale of this good result is that the model update, which is the slowest part of the model construction, is performed only when the model differs from the real system.

Chapter 5

Package injection

In this chapter we describe the proposed package injection approach which takes into account both the static data of packages and the contained maintainer scripts (see Figure 5.1). In this respect, package injectors have two main components each devoted to the management of the different kind of information to be injected. As discussed in the previous chapter, the system configuration injection already takes into account the static description of the packages. The other part to be considered in order to have a complete package injection is the injection of the maintainer scripts associated to the packages. This task requires specialized techniques and tools that are described in Section 5.1. As shown on right-hand side of Figure 5.1 the outcome of the overall package injection procedure consists of modeling elements which represent the considered package together with the contained scripts written as statements of the DSL presented in the Deliverable D3.2 and briefly recalled in Section 5.2. The details about the maintainer scripts injection are given in Section 5.3.

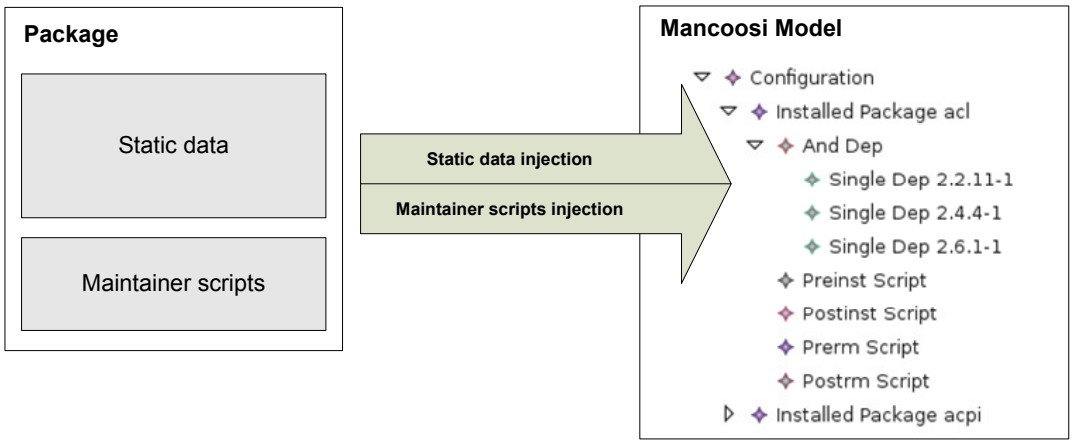

Figure 5.1: Overview of the package injection procedure

5.1 Gra2MoL: A domain specific language for extracting models from source code

Gra2MoL [CIM09] is a language especially tailored to address the problem of model extraction from source code. The idea of the extraction process is to automatically generate models conforming to a target metamodel from source code conforming to a specific grammar (see Figure 5.2).

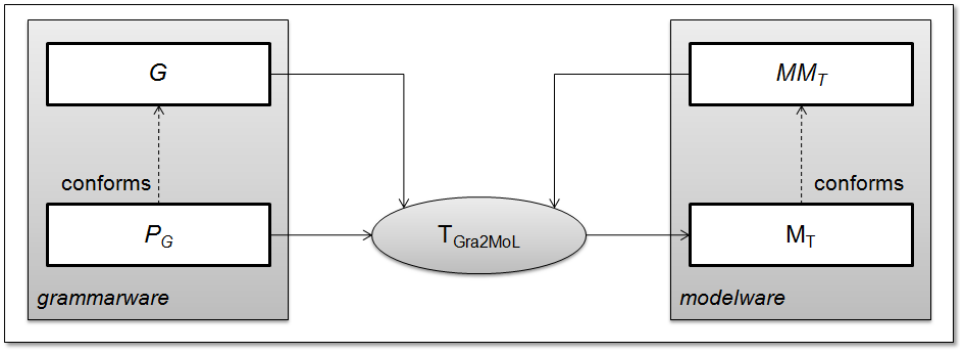

Figure 5.2: Overview of the Gra2Mol approach

Gra2MoL helps this process since it is a Domain Specific Language (DSL) that allows the specification of mappings between grammar elements and target metamodel elements. A Gra2MoL transformation definition consists of rules which transform grammar elements into model elements. Then in Gra2Mol the source element of a transformation rule is a grammar element rather than a metamodel element. A transformation rule is composed of four parts:

- ***from***: this part specifies a non terminal-symbol of the grammar and it is used to define the tree nodes where the rule is applied. This part can also include query operations to check if the nodes whose type is the non-terminal satisfy a defined structure;
- ***to***: this part specifies the type of the target elements which have to be generated when the rule is applied;
- ***queries***: this part is used to specify a set of query expressions defined in an OCL-like language. Queries are particularly useful in Gra2Mol since grammars and models have different nature. While programming language grammars produce syntax trees where references between elements are implicitly realized by means of identifiers, models are graphs where references between elements are explicit. Therefore, grammar-to-model transformations require intensive queries over the whole syntax tree to retrieve information that are out of the scope of the current rule;
- ***mapping***: the mapping is used to define the mappings between source and target elements. It contains a binding construct that can be used to establish the relationship between a source grammar element and a target metamodel element. A binding is written as an assignment using the operator "=", where the left-hand side must be a property of the target element metaclass and the right-hand side can be the variable specified in the from part of the rule, a query identifier, or a literal value. If it is a literal value, the value is directly assigned to the property of the left-hand side. If it is a query identifier, then the query is executed; the query execution may involve the execution of other rules. If

it is an expression, it is evaluated and two situations may arise: (i) the result is a node whose type corresponds to a terminal and in this case the result is directly assigned, or (ii) a rule to resolve the binding is executed.

Example 4 *Generating KDM model from Java code:* In the following we present a sample application of Gra2Mol borrowed by [CIM09] consisting of the automatic generation of KDM[1] models from Java code. In particular, KDM is a metamodel for knowledge discovery in software and defines a common vocabulary of knowledge related to software engineering artifacts, regardless of the implementation programming language and runtime platform. KDM is designed by the Object Management Group's foundation for software modernization. For more information, the interested reader can refer to the official Architecture-Driven Modernization (ADM) Website[2]. The main elements of the KDM metamodel are depicted in Figure 5.3: the root metaclass is `Segment` which contains different code models each composed of class units. Among the code elements of a class unit there are instances of the metaclass `methodUnit`.

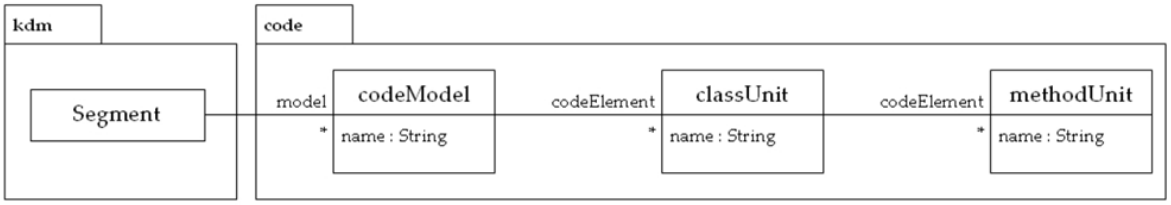

Figure 5.3: Sample KDM metamodel

According to Figure 5.2, to generate KDM models from source code, it is necessary the availability of the source grammar. A fragment of the ANTLR grammar of Java is reported in Figure 5.4. It is considered for specifying the Java to KDM transformation reported in Figure 5.5. For instance, the `createSegment` rule generates a target `Segment` instance from a source Java compilation unit. The query `class` retrieves the contained normal class declaration which will be used to set the reference `codeElement` of the generated element `model`.

5.2 The Mancoosi DSL

The Mancoosi DSL is a language which has been defined for specifying and to simulate the behavior of the maintainer scripts which are executed during package upgrades. The language has been conceived by analyzing existing maintainer scripts and identifying common recurrences which have been formalized in terms of specific metaclasses. The Mancoosi DSL consists of an abstract and concrete syntax, and of a semantics definition which has been given in an operational fashion by means of ATL transformations which change the source system configuration model and generate a target one according to the semantics of the executed DSL statements. The Mancoosi DSL is described in the Deliverable D3.2 and here we recall the main aspects which are useful to understand the maintainer scripts injection. The abstract syntax of the language is defined in terms of the Mancoosi metamodel, a small fragment of it is reported in

[1] Knowledge Discovery Metamodel (KDM): `http://www.omg.org/technology/kdm/index.htm`
[2] Architecture-Driven Modernization (ADM) : `http://adm.omg.org/`

```
compilationUnit
 : annotations? packageDeclaration?
   importDeclaration* typeDeclaration*
 ;

typeDeclaration
 : classOrInterfaceDeclaration
 ;

classOrInterfaceDeclaration
 : modifier* (classDeclaration | ...)
 ;

classDeclaration
 : normalClassDeclaration
 | ...
 ;

normalClassDeclaration
 : 'class' classId=Identifier (typeParameters)?
   ('extends' type)?
   ('implements' typeList)?
   classBody
 ;
```

```
classBody
 : '{' classBodyDeclaration* '}'
 ;

classBodyDeclaration
 : modifier* memberDecl
 | ...
 ;

memberDecl
 : methodDeclaration
 | ...
 ;

methodDeclaration
 : type methodName=Identifier
   methodDeclaratorRest
 ;

...
```

Figure 5.4: Fragment of the Java grammar

Figure 5.6: the InstalledPackage metaclass composes a number of scripts which consists of statements which can be both shell commands and templates like PostinstUdev and PostinstDesktop.

In the rest of the chapter we propose an injection approach which is able to represent in terms of models conforming to the Mancoosi metamodel, the maintainer scripts of the package to be installed or upgraded.

5.3 Maintainer script injection

The proposed approach to inject maintainer scripts relies on the Gra2Mol framework as outlined above. The injection process is depicted in Figure 5.7: G_{MS} is the grammar we defined for parsing the maintainer scripts. By means of ANTLR[3] we produce 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.

The grammar that we defined for the maintainer scripts is shown in Appendix. It is important to note that this grammar embodies ad-hoc productions corresponding to the identified recurrent templates. One assumption that we made is that the scripts that we want to parse are syntactically correct. This assumption is reasonable since we parse maintainer scripts that are integrant part of a Linux distribution.

The structure of the defined grammar is inspired by the Open Group Base specification for the Shell Command Language[4] and by the grammar of the Mancoosi DSL described in Deliverable D3.2. As shown in Listing 5.1 the grammar is able to recognize both the templates and shell

[3] ANTLR: http://www.antlr.org

[4] Shell Command Language: http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html

```
rule 'createSegment'
  from compilationUnit cu
  to kdm::Segment
  queries
    class : /cu//#normalClassDeclaration;
  mapping
    model = new code::CodeModel;
    model.name = "codeModel";
    model.codeElement = class;
end_rule

rule 'createClass'
  from normalClassDeclaration nc
  to code::ClassUnit
  queries
    ms : /nc//#methodDeclaration[@methodName.exists];
  mapping
    name = nc.classId;
    codeElement = ms;
end_rule

rule 'createMethod'
  from methodDeclaration md
  to code::MethodUnit
  queries
  mapping
    name = md.methodName;
end_rule
```

Figure 5.5: Sample Gra2Mol transformation

commands that cannot be recognized as a template. Moreover, the reported grammar is specific for Debian-based distributions. Suitable grammars should be defined for other distributions according to the common recurrences which have to be captured in terms of corresponding templates.

According to the grammar in Listing 5.1 each maintainer script starts with an header such as $\#!/bin/sh$. Therefore, the mainRule is composed of an header and a sequence of statements. A statement could be a `template`, one of the 52 found for Debian in Deliverable D2.1, a `command_statement`, a `control_statement`, a `loop_statement`, and finally a `redirection_statement`.

Listing 5.1: Main rules of the grammar

```
1  mainRule
2      : header (statement)*
3      ;
4
5  header
6      : SHARP EXCL path  (param)? '\n'
7      ;
8
9  statement
10     : template
11     | command_statement
12     | control_statement
13     | loop_statement
14     | redirection_statement
15     ;
16 ...
17 template :
18     ...
19     | templatePostinstDesktop
20     | ...
21     ;
22 ...
23 templatePostinstDesktop
24     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
         ↪? RSBRACK ('\n')? AND ('\n')? 'which' 'update-desktop-database' GREAT path
         ↪DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'update-desktop-database' param ('\n' |
```

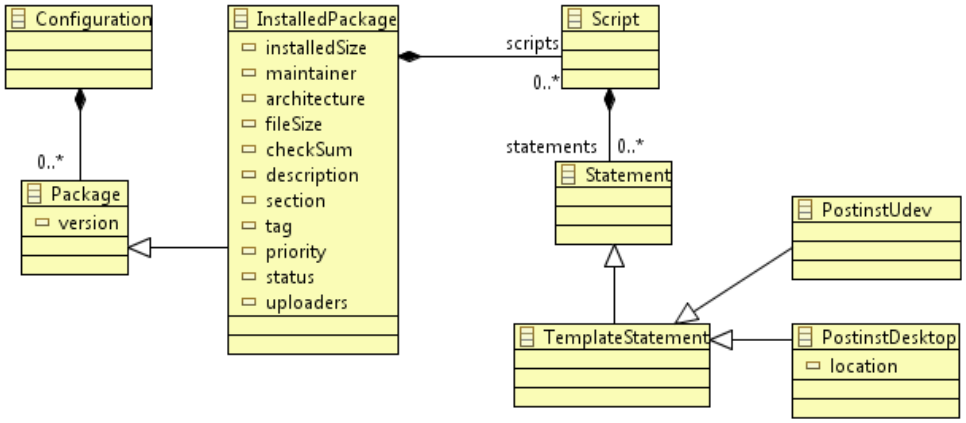

Figure 5.6: Fragment of the Package metamodel

```
        ↪ ';') 'fi' ('\n' | ';')
25    ;
26 ...
```

The non-terminal templatePostinstDesktop corresponds to the bash code reported in Listing 5.2 which is an identified Debian template as discussed in the Deliverable D2.1.

Listing 5.2: `postinst-desktop` Debian template

```
1 #!/bin/sh
2 if [ "$1" = "configure" ] && which update-desktop-database >/dev/null 2>&1 ; then
3     update-desktop-database -q
4 fi
```

The grammar reported in Listing 5.1 has been developed by using ANTLRWorks, the ANTLR GUI Development Environment[5]: Figure 5.8 shows a screenshot of the environment at work.

As shown in Figure 5.7, the grammar definition is the first step which is required to adopt

[5] ANTLRWorks: http://www.antlr.org/works/index.html

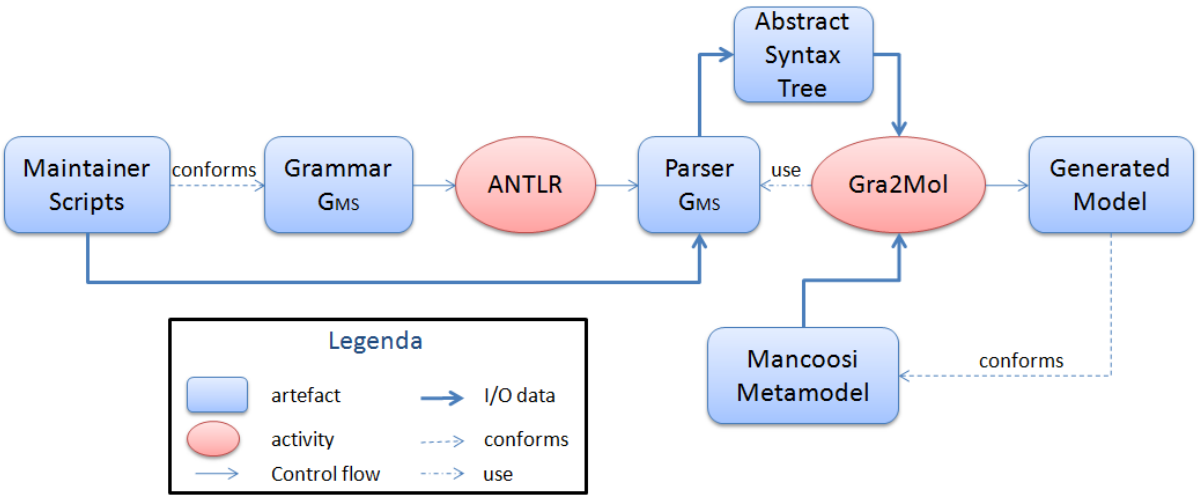

Figure 5.7: Maintain Scripts Injection

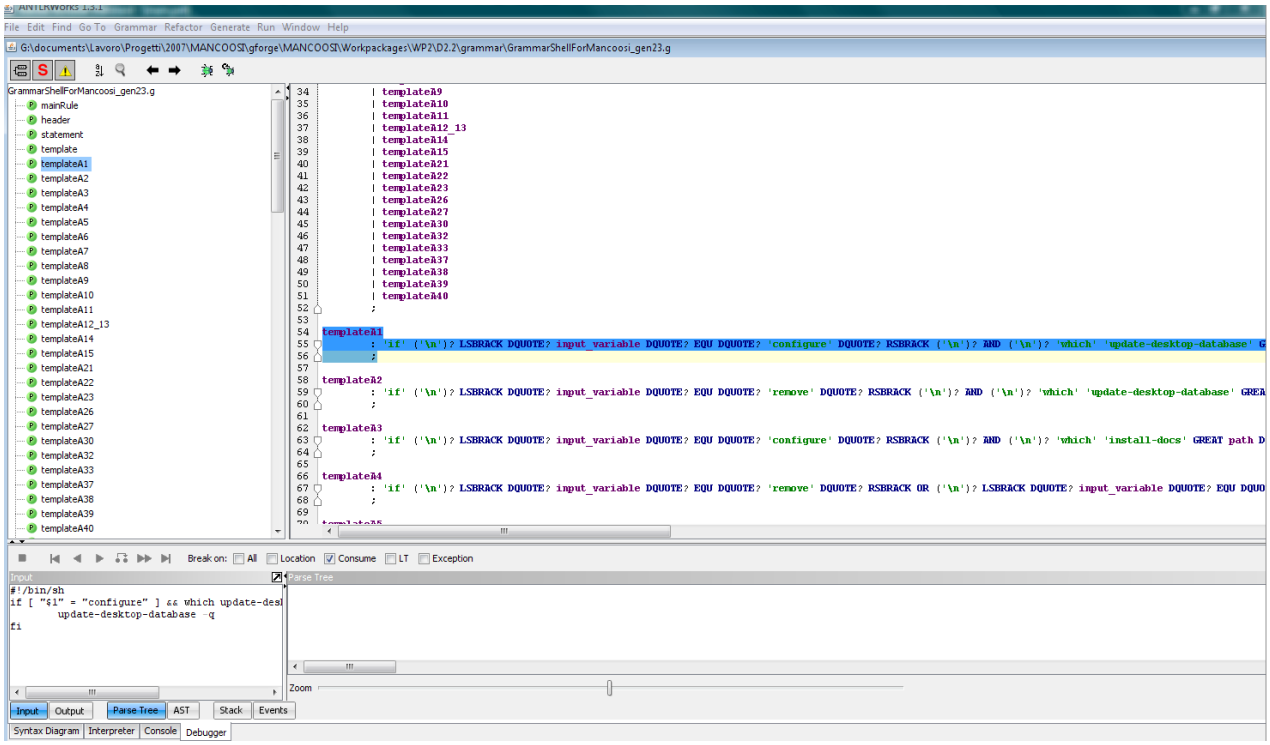

Figure 5.8: ANTLRWorks

Gra2Mol for the script injection. Then transformation rules have to be defined in order to establish how the elements in the abstract syntax tree which has been obtained by parsing the source code, have to be transformed in target model elements. Such transformation rules are specified by using the Gra2Mol language as in Listing 5.3. The reported transformation fragment consists of the following rules:

- mapInstalledPackage, this rule generates in the target model an instance of the InstalledPackage metaclass for each source file in a directory which contains all the maintainer scripts which have to be injected;
- mapScriptDefinition, this rule generates in the target model an instance of the metaclass Script for each mainRule element identified in the source code;
- mapTemplatePostinstDesktop and mapTemplatePostinstUdev generate target template elements with respect to the occurred template in the source code.

The specification order of the rules matters and rules are linked each others by means of the mappings directive. For instance, the rule mapScriptDefinition is executed for all the mains elements which are retrieved by the query in the mapInstalledPackage rule (see line 5 ofListing 5.3). The mapping at line 7 of Listing 5.3 sets the reference Script of the generated InstalledPackage with the element mains which contains all the elements generated by the rule mapScriptDefinition.

Listing 5.3: Excerpt of the Gra2Mol transformation for injecting maintainer scripts

```
1 rule 'mapInstalledPackage'
2     from file f
3     to InstalledPackage
4     queries
```

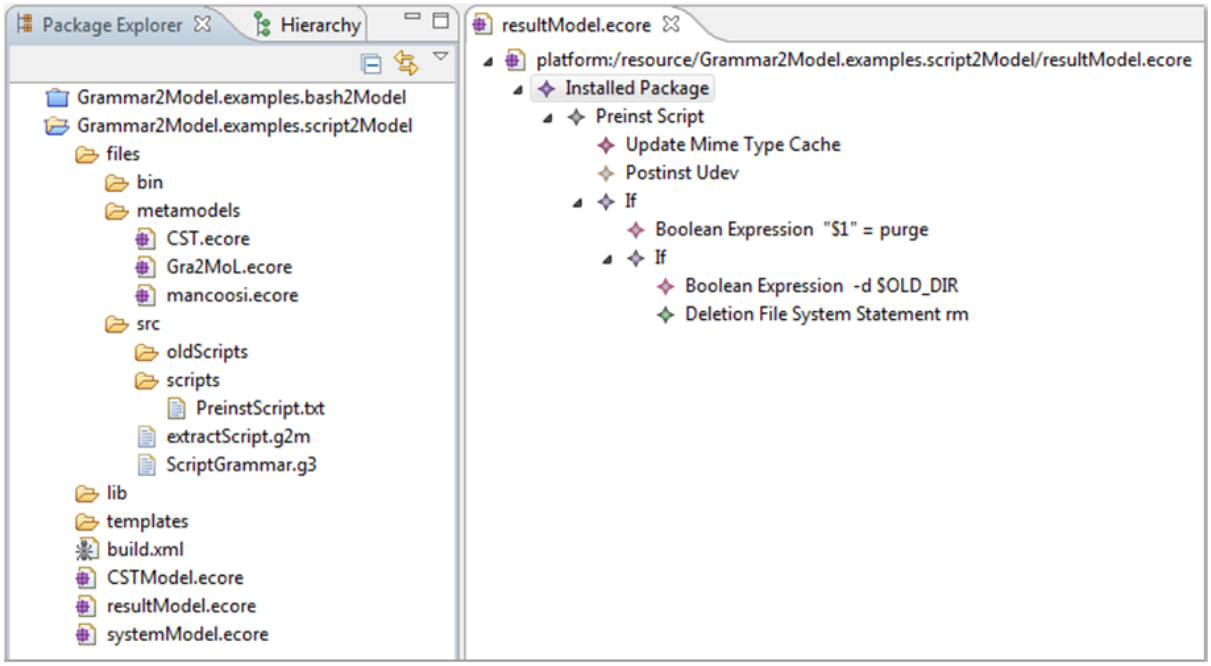

Figure 5.9: Sample injected maintainer scripts

```
        mains : //#mainRule;
    mappings
        Script = mains;
end_rule

rule 'mapScriptDefinition'
    from mainRule mr
    to Script
    queries
        stats : /mr//#command;
    mappings
        statements = stats;
end_rule

rule 'mapTemplatePostinstDesktop'
    from command/template//templatePostinstDesktop st
    to PostinstDesktop
    queries
        ...
    mappings
        ...
end_rule

rule 'mapTemplatePostinstUdev'
    from command/template//templatePostinstUdev st
    to PostinstUdev
    queries
        ...
    mappings
        ...
end_rule
```

A sample model which has been generated by means of the maintainer script injection approach previously presented is reported in Figure 5.9. The model consists of two DSL statements (i.e. `UpdateMimeTypeCache` and `PostinstUdev`) and of an `if` statement which does not correspond to any template, thus it has been represented in the generated model without manipulations.

Chapter 6

Failure detection

Once the model has been built, this opens to new static analysis scenarios. In order to understand the analysis that we can perform just using the model and without simulating upgrades it is useful to recall the classification of failures that we described in Deliverable D3.2. Let us recall also that the upgrade of a FOSS system typically requires the following main steps:

1. a user selects the package he/she aims to install or uninstall;
2. suitable dependencies resolution algorithms compute additional packages that are involved in the upgrade;
3. pre and post installation scripts are executed to perform the installation. If the required packages are available and the dependencies resolution algorithms resolve each dependency, then the step can be performed.

In Section 6.1 we devise a classification of possible failures which can occur during upgrade scenarios. We highlight those which are currently most difficult to manage since raised by incomplete or incorrect maintainer scripts. After this general classification, in Section 6.2 we discuss some failures which can be detected by means of static analysis on the considered system configuration model.

6.1 Failure Classification

The first failure classification can be provided by taking into account the time when failures are noticed. In this respect we distinguish among:

- ***Failure before the package commences installation***, upgrades can fail before the real installation of packages. Hence the system configuration is not changed. Typical situations that can occur are: (i) package lists that, either locally or remotely, have errors; (ii) package management database is locked or inaccessible; (iii) package management database is corrupted, incoherent or is in an erroneous state; (iv) dependencies may be missing due to a broken or unsynchronised repository; and (v) packages may refer to a package that has been deprecated and/or removed;
- ***Failure of scripts during upgrades***, these kinds of failures rely on the state of resources and for this reason it is difficult to simulate or predict. The main causes of these failures

can be: (i) scripts may try to access to non-existent files; (ii) scripts may have problems with access rights to/from files; (iii) scripts referred to may fail e.g. APT-Lua; (iv) pre and post install scripts may have insufficient permissions, etc.;

- ***Non valid configurations are reached***, the upgrade process reaches the end but the obtained configuration is "not valid" since, for instance, there are configuration files which refer to others which no longer exist in the file system;
- ***Slow failures*** (also named *Undetected failures* in [DCZT08]), even in this case, if the upgrade process reaches the end there is a chance that the obtained configuration contains failures that might not become apparent until another package is released or that it is in a generally unused part of the package and is "logic bomb";

In addition to this classification we can experience also other kinds of failures that typically occur during system upgrades. For a complete description of these aspects we refer to Deliverable D3.2.

6.2 Static Analysis

In this section we refer to failure that can be discovered before the package installation. In particular, in this section we show how by means of the model we are able to discover some errors that current package managers are not able to discover.

- *Discovering implicit dependencies among packages*: by means of the models that provide a common representation of the system's concepts we are able to discover dependencies that are implicit, i.e. dependencies that are not declared into the package's meta-information. For instance by analyzing the configuration files we can discover dependencies that configuration files of a package have with configuration files of another package. Since the configuration files of a package are obviously related to the package they belong to, a dependency among the two involved package should exist. However, the existence of this relation is not obvious since they are added by hand by maintainers.

  Actually, at the state of the art it is not possible to completely automate the extraction of these dependencies since the configuration files are interpreted by other programs that know the real semantics of what is written in these files. As future research direction we could imagine to define a domain specific language for writing configuration files. Another possible future research direction is to rely on a social and collaborative network managed by the distributions and shared among distribution maintainers, users, researchers.

  In the following we report some examples of implicit dependencies discovered on an Ubuntu 9.10 distribution by analyzing the configuration files by means of some defined heuristic:

  1. Even though no dependency is defined between the package `gdm` and the package `usplash`, our analysis raises that the configuration file `/etc/init.d/usplash` of the package `usplash` is required in the configuration file `/etc/init.d/gdm` of the package `gdm`. Listing 6.1 an extract of the configuration file showing this implicit dependency.

     Listing 6.1: Fragment of the configuration file `/etc/init.d/gdm`

```
1 ... if pidof usplash > /dev/null; then
2     usplash=:
3     orig_console="$(fgconsole)"
```

```
4       DO_NOT_SWITCH_VT=yes /etc/init.d/usplash start
5       # We've just shut down usplash, so don't log
6       # success as it will look weird on the console.
7       log_end_msg=:
8   else
9       usplash=false
10      log_end_msg=log_end_msg
11  fi
12  ...
```

The rationale of this part of this configuration file is: if `usplash` is running then `/etc/init.d/usplash` is invoked. The underlining assumption is that if `usplash` is running then the `usplash` package is installed. This assumption may easily turn to be false. For instance a package different from `usplash` and without any relation with this package could have simply executed a process coincidentally called `usplash`.

2. Even though no dependency is defined between the package `x11-common` and the package `gdm`, our analysis raises that the configuration file `/etc/init.d/gdm` of the package `gdm` could be required by the configuration file `/etc/gdm/failsafeXinit` of the package `x11-common`. The configuration file `/etc/init.d/gdm` may be invoked with a parameter `with_gdm`. The invocation of the file depends on a parameter that can be set to true. This is a case of "potential" dependency and it is clear that the responsibility to correctly set the parameter and to correctly manage this implicit dependency is delegated to the user.
3. Even though no dependency is defined between the package `gdm` and the package `x11-common`, the configuration file `/etc/X11/Xsession.options` of the package `x11-common` could be required by the configuration file `/etc/gdm/Xsession` of the package `gdm`. Actually the file in mentioned in a variable of the script and the variable is never used. However, it could be dangerous to have this variable defined since this could suggest to use it.
4. Let us consider the package `laptop-mode-tools`. This package does not have dependencies with the package `xset` even though the configuration file `/etc/laptop-mode/laptop-mode.conf` gives the possibility to the laptop mode tools of controlling the X display and the configuration file explicitly wrote that:

   > "*Using these settings, you can let laptop mode tools control the X display standby timeouts. This requires that you have installed the "xset" utility. It is part of the X.org server distribution and included in the package xorg-server-utils.*".

   Moreover the same configuration file recommends:

   > "*The X settings are not automatically applied on login, and this is impossible fix for the user, since laptop mode tools must operate as root. Therefore, it is recommended to add the following line to /etc/X11/Xsession as well: /usr/sbin/laptop_mode force*"

   Now let us imagine that we add this package and then we remove the package `laptop-mode-tools`. Then, since we modified the configuration file /etc/X11/Xsession of the `X11-common` package and since no dependency is defined between `X11-common` and `laptop-mode-tools`, the system configuration would have problems.

- *Discovering missing configuration files*: according to the system configuration model, some configuration files are required but they are not available in the system. For instance in Figure 6.1 the file /etc/ldap/ldap.conf is required by the installed package `libldap-2.4-2` but it is actually missing;

- *Discovering Mime-type dangling handlers*: according to the available information, the considered system should be able to manage a mime type, but the corresponding handler is missing in the system. For instance, in the configuration considered in Figure 6.1 the executable "dia" handling the mime type "application/x-dia-diagram" is missing even though according to the system configuration it appears to be a manageable mime type;
- *Discovering missing services*: the `init.d` file contains services that should start at the system start-up; some of the services would be not present in the system's configuration. By querying the configuration model, it is possible to detect such missing services.

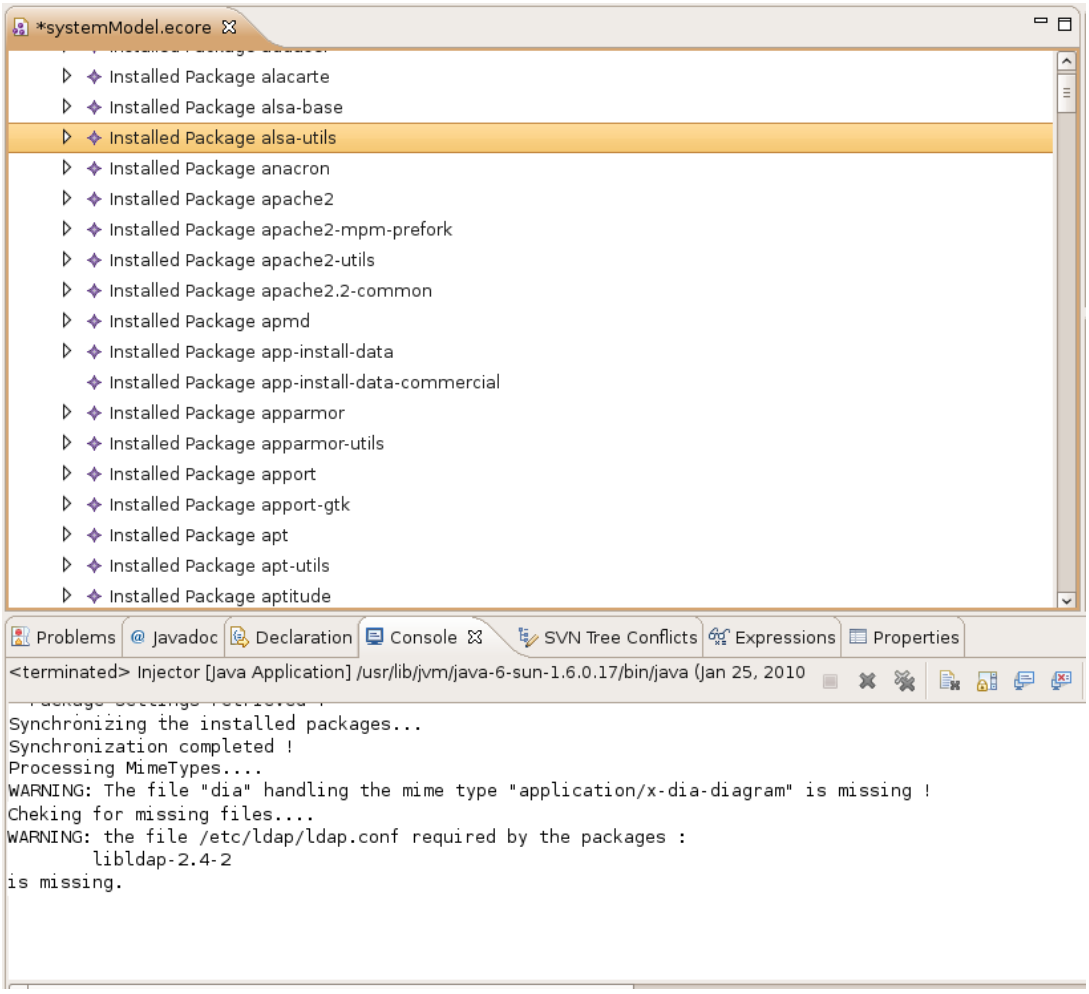

Figure 6.1: Some failures detected during system configuration injection

Many other failures can be detected by extending the set of managers presented in the previous chapters. Moreover, in the Deliverable 2.3, we will introduce also the support for the detection of failures which can occur during the simulation of package upgrades.

Chapter 7

Conclusion

In this deliverable we have described the instantiation of the Mancoosi metamodel on systems running the Ubuntu 9.10 Linux distribution. The instantiation has been performed by means of automated tools that have been conceived to be as distribution-independent as possible. For this reason the architecture of the model injection is layered. More precisely it consists of three layers:

- the lower layer is called *Mancoosi Model Management* and it is automatically and completely generated by the Eclipse Modeling Framework, which is the technology we used to implement the overall Mancoosi model injection. The *Mancoosi Model Management* layer provides the API to build the models conforming the Mancoosi metamodels;
- the middle layer is called *Mancoosi model injection infrastructure* and contains ten managers each devoted to the extraction of a particular aspect of the Linux running system, such as Configuration, Environment, Package, File System, etc. It is important to note that also this layer is completely distribution-independent. Moreover, this layer can be extended to cover more classes of errors during the upgrade simulation;
- the upper layer is called *Distribution-dependent Model Injector* and contains a model injector specialized for each distribution. Intuitively this layer contains instruments for querying the running system and for collecting useful information. The model is build by means of the API provided by the *Mancoosi model injection infrastructure* and then of the API provided by the lower layer.

The defined injection approach is used not only to build a first model of our running system but it is also used to keep the model synchronized with the running system. The synchronization is very useful to ensure that the model will not become obsolete even when a user intentionally or unintentionally makes a modification on the system without using typical upgrade instruments.

In Deliverable D2.3 we will show how the Mancoosi simulation will animate the models presented in this deliverable. The same models will be used also by Deliverable D3.3 to show how they can help the roll-back of an undesired upgrade.

Maintainers scripts grammar

```
grammar GrammarShellForMancoosi;

options {
    backtrack=true;
    k=1;
}


mainRule
    : header (statement)*
    ;

header
    : SHARP EXCL path  (param)? '\n'
    ;


statement
    : template
    | command_statement
    | control_statement
    | loop_statement
    | redirection_statement
    ;

template
    : templateA1
    | templateA2
    | templateA3
    | templateA4
    | templateA5
    | templateA6
    | templateA7
    | templateA8
    | templateA9
    | templateA10
    | templateA11
    | templateA12_13
    | templateA14
    | templateA15
    | templateA16
    | templateA17
    | templateA18
    | templateA19
    | templateA20
    | templateA21
    | templateA22
    | templateA23
    | templateA24
    | templateA25
    | templateA26
    | templateA27
    | templateA28
    | templateA29
    | templateA30
```

```
56      | templateA31
57      | templateA32
58      | templateA33
59      | templateA34
60      | templateA35
61      | templateA36
62      | templateA37
63      | templateA38
64      | templateA39
65      | templateA40
66      | templateA41
67      | templateA42
68      | templateA43
69      | templateA44
70      | templateA45
71      | templateA46
72      | templateA47
73      | templateA48
74      | templateA49
75      | templateA50
76      | templateA51
77      | templateA52
78      ;
79
80  templateA1
81      : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? 'which' 'update-desktop-database' GREAT SLASH '
            ↪dev' SLASH 'null' DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'update-desktop-
            ↪database' MINUS 'q' ('\n' | ';') 'fi' ('\n' | ';')
82      ;
83
84  templateA2
85      : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK ('\n')? AND ('\n')? 'which' 'update-desktop-database' GREAT SLASH 'dev'
            ↪ SLASH 'null' DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'update-desktop-database'
            ↪ MINUS 'q' ('\n' | ';') 'fi' ('\n' | ';')
86      ;
87
88  templateA3
89      : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? 'which' 'install-docs' GREAT SLASH 'dev' SLASH '
            ↪null' DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'install-docs' MINUS 'i' SLASH '
            ↪usr' SLASH 'share' SLASH 'doc-base' SLASH doc_id ('\n' | ';') 'fi' ('\n' | ';')
90      ;
91
92  templateA4
93      : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK OR ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'upgrade'
            ↪ DQUOTE? RSBRACK ('\n')? AND ('\n')? 'which' 'install-docs' GREAT SLASH 'dev'
            ↪SLASH 'null'
94      DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'install-docs' MINUS 'r' doc_id ('\n' | ';')
            ↪ 'fi' ('\n' | ';')
95      ;
96
97  templateA5
98      : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? LSBRACK MINUS 'x' SLASH 'usr' SLASH 'lib' SLASH '
            ↪emacsen-common' SLASH 'emacs-package-install' RSBRACK ';' 'then' ('\n')?
99      SLASH 'usr' SLASH 'lib' SLASH 'emacsen-common' SLASH 'emacs-package-install' pack
            ↪('\n' | ';') 'fi' ('\n' | ';')
100     ;
101
102 templateA6
103     :  'if' ('\n')? LSBRACK MINUS 'x' SLASH 'usr' SLASH 'lib' SLASH 'emacsen-common'
            ↪SLASH 'emacs-package-remove' RSBRACK ';' 'then' ('\n')?  SLASH 'usr' SLASH 'lib
            ↪' SLASH 'emacsen-common' SLASH 'emacs-package-remove' pack ('\n' | ';') 'fi'
            ↪('\n' | ';')
104     ;
105
106 templateA7
```

```
107     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? 'which' 'update-gconf-defaults' GREAT SLASH 'dev'
            ↪ SLASH 'null' DIGIT GREATAND DIGIT ';' 'then' ('\n')? 'update-gconf-defaults'
            ↪('\n' | ';') 'fi' ('\n' | ';')
108     ;
109
110 templateA8
111     : 'if' ('\n')?  'which' 'update-gconf-defaults' GREAT SLASH 'dev' SLASH 'null' DIGIT
            ↪ GREATAND DIGIT ';' 'then' ('\n')? 'update-gconf-defaults' ('\n' | ';') 'fi'
            ↪('\n' | ';')
112     ;
113
114 templateA9
115     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK  ';' 'then' ('\n')?  'gconf-schemas' DMINUS 'register' schemas ('\n'
            ↪| ';') 'fi' ('\n' | ';')
116     ;
117
118 templateA10
119     :  'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK OR ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'upgrade'
            ↪ DQUOTE? RSBRACK ';' 'then' ('\n')?  'gconf-schemas' DMINUS 'unregister'
            ↪schemas ('\n' | ';') 'fi' ('\n' | ';')
120     ;
121
122 templateA11
123     :  'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'purge' DQUOTE?
            ↪RSBRACK  ';' 'then' ('\n')? 'OLD_DIR' EQU  SLASH 'etc'  SLASH 'gconf'  SLASH '
            ↪schemas' ('\n')? 'SCHEMA_FILES' EQU DQUOTE schemas DQUOTE '\n' 'if' ('\n')?
124     LSBRACK ('\n')? MINUS 'd' DOLLAR 'OLD_DIR' RSBRACK ';' 'then' ('\n')? 'for' ('\n')?
            ↪'SCHEMA' ('\n')? 'in' ('\n')? DOLLAR 'SCHEMA_FILES' ';' 'do' '\n'  'rm' MINUS '
            ↪f' DOLLAR 'OLD_DIR' SLASH DOLLAR 'SCHEMA' '\n' 'done' ('\n' | ';')  'rmdir'
            ↪MINUS 'p'
125     DMINUS 'ignore-fail-on-non-empty' DOLLAR 'OLD_DIR' ('\n' | ';') 'fi' ('\n' | ';') '
            ↪fi' ('\n' | ';')
126     ;
127
128 templateA12_13
129     : 'if' ('\n')?  'which' 'update-icon-caches' GREAT SLASH 'dev' SLASH 'null' DIGIT
            ↪GREATAND DIGIT ';' 'then' ('\n')? 'update-icon-caches' dirlist ('\n' | ';') 'fi
            ↪' ('\n' | ';')
130     ;
131
132 templateA14
133     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK  ';' 'then' ('\n')? 'install-info' DMINUS 'quiet' file ('\n' | ';') '
            ↪fi' ('\n' | ';')
134     ;
135
136 templateA15
137     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK ('\n')? OR ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? '
            ↪upgrade' DQUOTE? RSBRACK ';'
138     'then' ('\n')? 'install-info' DMINUS 'quiet' DMINUS 'remove' file ('\n' | ';') 'fi'
            ↪('\n' | ';')
139     ;
140 //
141 templateA16
142     :  'if' ('\n')? LSBRACK param DQUOTE SLASH 'etc' SLASH 'init.d' SLASH script DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'update-rc.d' script initparms GREAT path  OR
            ↪simple_command ('\n' | ';') 'fi' ('\n' | ';')
143     ;
144
145 templateA17
146     : 'if' ('\n')? LSBRACK param DQUOTE SLASH 'etc' SLASH 'init.d' SLASH word DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'update-rc.d' word param+ GREAT path ('\n' | ';')
147     'if' ('\n')? LSBRACK param DQUOTE input_variable DQUOTE RSBRACK ';' 'then' ('\n')? '
            ↪_dh_action' EQU 'restart'  ('\n')? 'else'  ('\n')?  '_dh_action' EQU 'start'
            ↪('\n' | ';') 'fi' ('\n' | ';')
```

```
148     'if' ('\n')? LSBRACK param DQUOTE AP  'which' 'invoke-rc.d' DIGIT GREAT path AP
            ↪DQUOTE RSBRACK ';' 'then' ('\n')?  'invoke-rc.d' word DOLLAR '_dh_action' OR
            ↪simple_command ('\n')?
149     'else' ('\n')? SLASH 'etc' SLASH 'init.d' SLASH word DOLLAR '_dh_action' OR
            ↪simple_command  ('\n' | ';') 'fi'  ('\n' | ';') 'fi' ('\n' | ';')
150     ;
151
152 templateA18
153     : 'if' ('\n')? LSBRACK param DQUOTE SLASH 'etc' SLASH 'init.d' SLASH word DQUOTE
            ↪RSBRACK AND  LSBRACK DQUOTE input_variable DQUOTE EQU 'remove' RSBRACK ';'
154     'then' ('\n')?  'if' ('\n')? LSBRACK param DQUOTE AP  'which' 'invoke-rc.d' DIGIT
            ↪GREAT path AP DQUOTE RSBRACK ';' 'then' ('\n')?  'invoke-rc.d' word 'stop' OR
            ↪simple_command ('\n')?
155     'else' ('\n')? SLASH 'etc' SLASH 'init.d' SLASH word 'stop' OR simple_command  ('\n'
            ↪ | ';') 'fi' ('\n' | ';') 'fi' ('\n' | ';')
156     ;
157
158 templateA19
159     :  'if' ('\n')? LSBRACK param DQUOTE SLASH 'etc' SLASH 'init.d' SLASH word DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'update-rc.d' word param+ GREAT path ('\n' | ';')
160     'if' ('\n')? LSBRACK param DQUOTE AP  'which' 'invoke-rc.d' DIGIT GREAT path AP
            ↪DQUOTE RSBRACK ';' 'then' ('\n')?  'invoke-rc.d' word 'start' OR simple_command
            ↪ ('\n')?
161     'else' ('\n')? SLASH 'etc' SLASH 'init.d' SLASH word 'start' OR simple_command  ('\n
            ↪' | ';') 'fi' ('\n' | ';') 'fi' ('\n' | ';')
162     ;
163
164 templateA20
165     :  'if' ('\n')? LSBRACK param DQUOTE SLASH 'etc' SLASH 'init.d' SLASH word DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'if' ('\n')? LSBRACK param DQUOTE AP  'which' '
            ↪invoke-rc.d' DIGIT GREAT path AP DQUOTE RSBRACK ';'
166     'then' ('\n')?  'invoke-rc.d' word 'stop' OR simple_command ('\n')? 'else' ('\n')?
            ↪SLASH 'etc' SLASH 'init.d' SLASH word 'stop' OR simple_command  ('\n' | ';') '
            ↪fi' ('\n' | ';') 'fi' ('\n' | ';')
167     ;
168
169 templateA21
170     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'purge' DQUOTE?
            ↪RSBRACK ';' 'then' ('\n')? 'update-rc.d' param 'remove' GREAT path OR
            ↪simple_command ('\n' | ';') 'fi' ('\n' | ';')
171     ;
172
173 templateA22
174     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ';' 'then' ('\n')? 'ldconfig' ('\n' | ';') 'fi' ('\n' | ';')
175     ;
176
177 templateA23
178     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK ';' 'then' ('\n')? 'ldconfig' ('\n' | ';') 'fi' ('\n' | ';')
179     ;
180
181 templateA24
182     : 'inst' EQU SLASH 'etc' SLASH 'menu-methods' SLASH word (';' | '\n') 'if'  LSBRACK
            ↪param DOLLAR 'inst' RSBRACK ';' 'then' ('\n')? 'chmod' 'a+x' DOLLAR 'inst' ('\n
            ↪' | ';')
183     'if'  LSBRACK param DQUOTE AP  'which' 'update-menus' DIGIT GREAT path AP DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'update-menus'  ('\n' | ';') 'fi' ('\n' | ';') 'fi'
            ↪('\n' | ';')
184     ;
185
186 templateA25
187     :  'inst' EQU SLASH 'etc' SLASH 'menu-methods' SLASH word (';' | '\n') 'if'  LSBRACK
            ↪ DQUOTE input_variable DQUOTE EQU DQUOTE 'remove' DQUOTE RSBRACK AND LSBRACK
            ↪param DQUOTE DOLLAR 'inst' DQUOTE RSBRACK ';'
188     'then' ('\n')?  'chmod' 'a-x' DOLLAR 'inst' ('\n' | ';') 'fi' ('\n' | ';')  'if'
            ↪LSBRACK param DQUOTE AP  'which' 'update-menus' DIGIT GREAT path AP DQUOTE
            ↪RSBRACK ';' 'then' ('\n')? 'update-menus'  ('\n' | ';') 'fi' ('\n' | ';')
189     ;
190
```

```
templateA26
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
        ↪? RSBRACK ('\n')? AND ('\n')? LSBRACK param DQUOTE AP 'which' 'update-menus'
        ↪DIGIT GREAT path AP DQUOTE RSBRACK ';'
    'then' ('\n')? 'update-menus' ('\n' | ';') 'fi' ('\n' | ';')
    ;

templateA27
    :  'if' ('\n')? LSBRACK param DQUOTE AP 'which' 'update-menus' DIGIT GREAT path AP
        ↪DQUOTE RSBRACK ';' 'then' ('\n')? 'update-menus' ('\n' | ';') 'fi' ('\n' | ';')
    ;

templateA28
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
        ↪? RSBRACK ';' 'then' ('\n')?  'if' ('\n')? LSBRACK param SLASH 'boot' SLASH '
        ↪System.map' MINUS word RSBRACK ';'
    'then'  'depmod' param param SLASH 'boot' SLASH 'System.map' MINUS word word OR  ('\
        ↪n' | ';') 'fi' ('\n' | ';') 'fi' ('\n' | ';')
    ;

templateA29
    :  'if' ('\n')? LSBRACK param SLASH 'boot' SLASH 'System.map' MINUS word RSBRACK ';'
        ↪ 'then'  'depmod' param param SLASH 'boot' SLASH 'System.map' MINUS word word
        ↪OR  ('\n' | ';') 'fi' ('\n' | ';')
    ;

templateA30
    : 'PYTHON' EQU number ('\n' | ';') 'if' ('\n')? 'which' DOLLAR 'PYTHON' GREAT path
        ↪DIGIT GREATAND DIGIT ('\n')? AND ('\n')?  LSBRACK param SLASH 'usr' SLASH 'lib'
        ↪ SLASH DOLLAR 'PYTHON' SLASH 'compileall.py' RSBRACK ';'
    'then' ('\n')? 'DIRLIST' EQU DQUOTE param+ DQUOTE ( '\n' | ';' ) 'for' ('\n')? word
        ↪'in' DOLLAR 'DIRLIST' ';' 'do' '\n' DOLLAR 'PYTHON' SLASH 'usr' SLASH 'lib'
        ↪SLASH DOLLAR 'PYTHON' SLASH 'compileall.py' param word ('\n' | ';') 'done' ('\n
        ↪' | ';') 'fi' ('\n' | ';')
    ;

templateA31
    : 'dpkg' param param VBAR ('\n')? 'awk' AP input_variable TILDE  SLASH BSLASH DOT '
        ↪py' DOLLAR SLASH LGRAF 'print' input_variable DQUOTE 'c' BSLASH 'n' DQUOTE
        ↪DQUOTE input_variable DQUOTE RGRAF AP VBAR ('\n')? 'xargs' 'rm' param GREATAND
        ↪DIGIT ('\n' | ';')
    ;

templateA32
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
        ↪? RSBRACK ('\n')? AND ('\n')? 'which' 'scrollkeeper-update' GREAT path DIGIT
        ↪GREATAND DIGIT ';' 'then' ('\n')? 'scrollkeeper-update' param ('\n' | ';') 'fi'
        ↪ ('\n' | ';')
    ;

templateA33
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
        ↪RSBRACK ('\n')? AND ('\n')? 'which' 'scrollkeeper-update' GREAT path DIGIT
        ↪GREATAND DIGIT ';' 'then' ('\n')? 'scrollkeeper-update' param ('\n' | ';') 'fi'
        ↪ ('\n' | ';')
    ;

 templateA34
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
        ↪? RSBRACK ';' 'then' ('\n')? 'rm' param param ('\n' | ';') 'for' 'ordcat' 'in'
        ↪word+ ';'
    'do' ('\n')? 'update-catalog' DMINUS 'quiet'  DMINUS 'add' param DOLLAR LGRAF '
        ↪ordcat' RGRAF ('\n' | ';') 'done' ('\n' | ';') 'update-catalog' DMINUS 'quiet'
        ↪DMINUS 'add' DMINUS 'super' param ('\n' | ';') 'fi' ('\n' | ';')
    ;

 templateA35
    : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
        ↪RSBRACK ('\n')? OR ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? '
        ↪upgrade' DQUOTE? RSBRACK ';'
```

```
233     'then' ('\n')? 'update-catalog' DMINUS 'quiet' DMINUS 'remove' DMINUS 'super' param
            ↪('\n' | ';') 'fi' ('\n' | ';')
234     ;
235
236  templateA36
237     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'purge' DQUOTE?
            ↪RSBRACK ('\n')? ';' 'then' ('\n')? 'rm' param param param ('\n' | ';') 'fi' ('\
            ↪n' | ';')
238     ;
239
240 templateA37
241     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? LSBRACK param DQUOTE AP 'which' 'update-mime-
            ↪database'  GREAT path AP DQUOTE RSBRACK ';'
242     'then' ('\n')? 'update-mime-database' SLASH 'usr' SLASH 'share' SLASH' mime' ('\n' |
            ↪ ';') 'fi' ('\n' | ';')
243     ;
244
245 templateA38
246     :  'if' ('\n')? LSBRACK param DQUOTE AP 'which' 'update-mime-database' DIGIT GREAT
            ↪path AP DQUOTE RSBRACK ';' 'then' ('\n')? 'update-mime-database' SLASH 'usr'
            ↪SLASH 'share' SLASH 'mime' ('\n' | ';') 'fi' ('\n' | ';')
247     ;
248
249 templateA39
250     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n')? AND ('\n')? LSBRACK param DQUOTE AP 'which' 'update-mime'
            ↪DIGIT GREAT path AP DQUOTE RSBRACK ';'
251     'then'  ('\n')? 'update-mime' ('\n' | ';') 'fi' ('\n' | ';')
252     ;
253
254 templateA40
255     :  'if' ('\n')?  'which' 'update-mime' GREAT path DIGIT GREATAND DIGIT ';' 'then'
            ↪('\n')? 'update-mime' param ('\n' | ';') 'fi' ('\n' | ';')
256     ;
257
258 templateA41
259     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n' | ';') 'then' ('\n')? 'if' ('\n')? 'which' 'suidregister' GREAT
            ↪ path DIGIT GREATAND DIGIT ('\n')? AND ('\n')? LSBRACK param SLASH 'etc' SLASH
            ↪'suid.conf' RSBRACK ('\n' | ';')
260     'then' ('\n')? 'suidregister' param param path param param param ('\n' | ';')
261     'elif' ('\n')? LSBRACK param param RSBRACK (';' |'\n') 'then' ('\n')? 'chown' param
            ↪COLON param param ('\n' | ';') 'chmod' param param ('\n' | ';') 'fi' ('\n' |
            ↪';') 'fi' ('\n' | ';')
262     ;
263
264 templateA42
265     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK ('\n')? AND ('\n')? LSBRACK param SLASH 'etc' SLASH 'suid.conf' RSBRACK
            ↪ ('\n')? AND ('\n')? 'which' 'suidunregister' GREAT path DIGIT GREATAND DIGIT
            ↪';' 'then' ('\n')? 'suidunregister' param param param ('\n' | ';') 'fi' ('\n' |
            ↪ ';')
266     ;
267
268
269 templateA43
270     :'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'install' DQUOTE?
            ↪RSBRACK OR ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'upgrade'
            ↪ DQUOTE? RSBRACK ';' 'then' ('\n')? 'if' ('\n')? LSBRACK param DQUOTE word
            ↪DQUOTE RSBRACK ';' 'then' ('\n')? 'if' ('\n')? LSBRACK
271      DQUOTE AP 'md5sum'  BSLASH DQUOTE word BSLASH DQUOTE VBAR 'sed' param BSLASH DQUOTE
            ↪ 's' SLASH DOT '*' SLASH SLASH BSLASH DQUOTE AP DQUOTE EQU DQUOTE AP 'dpkg-
            ↪query' param param EQU APSINGLE DOLLAR LGRAF 'Conffiles' RGRAF APSINGLE word
            ↪VBAR 'sed' param param BSLASH DQUOTE
272      BSLASH BSLASH BSLASH BSLASH APSINGLE word APSINGLE 's' SLASH DOT '*' SLASH SLASH 'p
            ↪' BSLASH DQUOTE AP DQUOTE RSBRACK ('\n')? 'then' ('\n')? 'rm' param DQUOTE
            ↪word DQUOTE ('\n' | ';') 'fi' ('\n' | ';') 'fi' ('\n' | ';') 'if'('\n')?
            ↪LSBRACK param DQUOTE word DQUOTE RSBRACK ';'
```

```
273       'then' ('\n')? 'rm' param DQUOTE word DQUOTE ('\n' | ';') 'fi' ('\n' | ';') 'fi'
              ↪('\n' | ';')
274     ;
275
276 templateA44
277     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK (';' | '\n') 'then' ('\n')? 'if' LSBRACK param DQUOTE word DQUOTE
            ↪RSBRACK (';' | '\n') 'then' ('\n')? 'echo' DQUOTE 'Preserving' 'user' 'changes
            ↪' 'to' word
278     DOT DOT DOT DQUOTE (';' | '\n') 'if' LSBRACK param DQUOTE word DQUOTE RSBRACK (';'
            ↪ | '\n') 'then' ('\n')? 'mv' param DQUOTE word DQUOTE DQUOTE word DQUOTE ('\n
            ↪' | ';') 'fi' ('\n' | ';') 'mv' param DQUOTE word DQUOTE DQUOTE word DQUOTE
            ↪('\n' | ';') 'fi' ('\n' | ';') 'fi' ('\n' | ';')
279     ;
280
281 templateA45
282     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK (';' | '\n') 'then' ('\n')? LPAR ('\n')? 'while' 'read' 'line' (';')
            ↪? 'do' ('\n')? 'set' DMINUS DOLLAR 'line' (';' | '\n') 'dir' EQU DQUOTE
            ↪input_variable DQUOTE ';' 'mode' EQU DQUOTE input_variable DQUOTE ';' 'user'
            ↪EQU DQUOTE input_variable DQUOTE ';' 'group' EQU DQUOTE input_variable DQUOTE
            ↪(';' | '\n') 'if' LSBRACK EXCL param DQUOTE DOLLAR 'dir' DQUOTE RSBRACK
283     (';' | '\n') 'then' ('\n')? 'if' 'mkdir' DQUOTE DOLLAR 'dir' DQUOTE DIGIT GREAT
            ↪path (';' | '\n') 'then' ('\n')? 'chown' DQUOTE DOLLAR 'user' DQUOTE COLON
            ↪DQUOTE DOLLAR 'group' DQUOTE DQUOTE DOLLAR 'dir' DQUOTE (';' | '\n') 'chmod'
            ↪DQUOTE DOLLAR 'mode' DQUOTE DQUOTE DOLLAR 'dir' DQUOTE ('\n' | ';') 'fi'('\n' |
            ↪ ';') 'fi' (';' | '\n') 'done' (';' | '\n') RPAR DLESS word ('\n' | ';' )
            ↪word ('\n' | ';') 'fi' ('\n' | ';')
284     ;
285
286 templateA46
287     : LPAR ('\n')? 'while' 'read' 'dir' (';')? 'do' ('\n') 'rmdir' DQUOTE DOLLAR 'dir'
            ↪DQUOTE DIGIT GREAT path OR 'true' ('\n' | ';') 'done' ('\n' | ';') RPAR DLESS
            ↪ word ('\n' | ';' ) param ('\n' | ';' ) word ('\n' | ';')
288     ;
289
290 templateA47
291     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n' | ';') 'then' ('\n')? 'update-alternatives' DMINUS 'install'
            ↪SLASH 'usr' SLASH 'bin' SLASH 'x-window-manager' 'x-window-manager' word param
            ↪ ('\n' | ';') 'fi' ('\n' | ';')
292     ;
293
294 templateA48
295     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'configure' DQUOTE
            ↪? RSBRACK ('\n' | ';') 'then' ('\n')? 'update-alternatives' DMINUS 'install'
            ↪SLASH 'usr' SLASH 'bin' SLASH 'x-window-manager' 'x-window-manager' word param
            ↪DMINUS 'slave' SLASH 'usr' SLASH 'share'
296     SLASH 'man' SLASH 'man1' SLASH 'x-window-manager.1.gz' 'x-window-manager.1.gz' param
            ↪ ('\n' | ';') 'fi' ('\n' | ';')
297     ;
298
299 templateA49
300     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'remove' DQUOTE?
            ↪RSBRACK ('\n' | ';') 'then' ('\n')? 'update-alternatives' DMINUS 'remove' 'x-
            ↪window-manager' param ('\n' | ';') 'fi' ('\n' | ';')
301     ;
302
303 templateA50
304     : 'if' ('\n')? 'which' 'update-fonts-dir' GREAT path DIGIT GREATAND DIGIT ';' 'then'
            ↪ ('\n')? command_statement+ 'fi' ('\n' | ';')
305     ;
306
307 templateA51
308     : 'if' LSBRACK param DQUOTE AP 'which' 'update-fonts-dir' DIGIT GREAT path AP DQUOTE
            ↪ RSBRACK ';' 'then' ('\n')? command_statement+ 'fi' ('\n' | ';')
309     ;
310
311 templateA52
```

```
312     : 'if' ('\n')? LSBRACK DQUOTE? input_variable DQUOTE? EQU DQUOTE? 'purge' DQUOTE?
          ↪RSBRACK ('\n')? AND ('\n')? LSBRACK param SLASH 'usr' SLASH 'share' SLASH '
          ↪debconf' SLASH 'confmodule' RSBRACK  ';' 'then' ('\n')? DOT SLASH 'usr' SLASH '
          ↪share' SLASH 'debconf' SLASH 'confmodule' ('\n' | ';') 'db_purge' ('\n' | ';')
          ↪'fi' ('\n' | ';')
313     ;
314
315 doc_id
316     : id
317     ;
318
319 pack
320     : id
321     ;
322
323 schemas
324     : word
325     ;
326
327 dirlist
328     : path+
329     ;
330
331 file
332     : id
333     ;
334
335 initparms
336     : param+
337     ;
338
339 script
340     : id
341     ;
342
343 redirection_statement
344     : statement_group redirection ('\n' | ';')
345     ;
346
347 redirection
348     : (GREAT
349     //| LESS
350     | DIGIT GREAT
351     //| DIGIT LESS
352     | DGREAT
353     //| DIGIT DGREAT
354     | DLESS
355     //| LESSAND
356     //| DIGIT LESSAND
357     | GREATAND
358     | DIGIT GREATAND
359     //| DLESSDASH
360     //| DIGIT DLESSDASH
361     //| LESSAND
362     //| DIGIT LESSAND
363     | GREATAND
364     | DIGIT GREATAND
365     //| DLESSDASH
366     //| DIGIT DLESSDASH
367     | ANDGREAT
368     //| DIGIT LESSGREAT
369     //| LESSGREAT
370     //| CLOBBER
371     //| DIGIT CLOBBER
372     //| DIGIT DLESS
373     ) ( path | word)
374     | LESSAND DIGIT
375     | DIGIT LESSAND DIGIT
376     | GREATAND DIGIT
377     | DIGIT GREATAND DIGIT
```

```
    //| GREATAND MINUS
    //| DIGIT GREATAND MINUS
    //| LESSAND MINUS
    //| DIGIT LESSAND MINUS
    ;

statement_group
    : LPAR ('\n')? statement* RPAR
    | LGRAF ('\n')? statement* (';')? RGRAF
    ;


command_statement
    : list_of_commands ('\n' | ';')
    ;

command_name
    : id
    |path
    |DOT
    ;

nestedCommand
    : command_name param*
    ;

control_statement
    : if_statement
    | case_statement
    ;

loop_statement
    : for_statement
    | until_statement
    | while_statement
    ;

until_statement
    : 'until' ('\n')? condition (';')? 'do' ('\n')? statement* 'done' ('\n' | ';')
    ;


while_statement
    : 'while' ('\n')? condition (';')? 'do' ('\n')? statement* 'done' ('\n' | ';')
    ;

case_statement
    : 'case' ('\n')? (DQUOTE)? word (DQUOTE)? 'in' ('\n')? body_case+ 'esac' ('\n' |
      ↪ ';')
    ;

body_case
    : word ('\n')? (VBAR word)* ('\n')? RPAR ('\n')? (statement)* ('\n')? DSEMI ('\n')?
    ;

if_statement
    : 'if' ('\n')? condition (';')? 'then' ('\n')? statement* ifelse_branch* else_branch
      ↪?  'fi' ('\n' | ';')
    ;

ifelse_branch
    : 'elif' ('\n')? condition (';')? 'then' ('\n')? statement*
    ;

else_branch
    : 'else' ('\n')? statement*
    ;

condition
    : conditional_expr_list ((AND | OR) command_expr_list)*
```

```
446     | command_expr_list ((AND | OR) conditional_expr_list)*
447     ;
448
449 conditional_expr_list
450     : conditional_expression ((AND | OR) ('\n')? conditional_expression)*
451     ;
452
453 command_expr_list
454     : pipeline ((AND | OR) ('\n')? pipeline)*
455     ;
456
457 for_statement
458     : 'for' ('\n')? word 'in' word (';')? 'do' ('\n')? statement* 'done' ('\n' | ';')
459     ;
460
461 assignment_statement
462     : word EQU param
463     | word EQU DQUOTE param+ DQUOTE
464     ;
465
466 conditional_expression
467     :  LSBRACK EXCL? (boolean_expression | command_expression)  RSBRACK
468     ;
469
470 boolean_expression
471     : DQUOTE? word DQUOTE? (EQU | NOTEQU) DQUOTE? (word | path) DQUOTE?
472     ;
473
474 command_expression
475     : param DQUOTE? (param)+ DQUOTE?
476     ;
477
478 substitution_variable
479     : DOLLAR LGRAF id RGRAF
480     ;
481
482 input_variable
483     : DOLLAR DIGIT
484     ;
485
486 in_command_variable
487     : DOLLAR id
488     ;
489
490 external_variable
491     : UNDERSCORE id
492     | DOLLAR UNDERSCORE id
493     ;
494
495 special_variable
496     : DOLLAR QMARK
497     | DOLLAR DOLLAR
498     | DOLLAR EXCL
499     ;
500
501 shell_variable
502     : DOLLAR 'HOME'
503     | DOLLAR 'PATH'
504     | DOLLAR 'PS' DIGIT
505     ;
506
507 simple_command
508     :  assignment_statement
509     | command_name (DQUOTE? param+ DQUOTE?)*
510     ;
511
512 param
513     : MINUS word
514     | DMINUS word?
515     | word
```

```
516     | path
517     | redirection
518     | COLON
519     | AP nestedCommand AP
520     | number
521     | command_name
522     | BSLASH
523     | SLASH
524     | APSINGLE
525     | TILDE
526     | '*'
527     ;
528
529 word
530     : input_variable
531     | in_command_variable
532     | shell_variable
533     | special_variable
534     | external_variable
535     | substitution_variable
536     | id
537     ;
538
539 pipeline
540     : ('time')? simple_command (VBAR ('\n')? simple_command)*
541     ;
542
543 list_of_commands
544     : pipeline ((AND | OR) pipeline)*
545     | asynchronous
546     ;
547
548 asynchronous
549     : pipeline (AMPERSAND pipeline)* (AMPERSAND)?
550     ;
551
552 path
553     : (DOLLAR)? (id)? (SLASH (DOLLAR)? id)+ SLASH?
554     ;
555
556  number : DIGIT+ ((DOT | MINUS) DIGIT+)*
557     ;
558
559 id  : 'update-desktop-database'
560     | 'which'
561     | 'configure'
562     | 'remove'
563     | 'install-docs'
564     | 'upgrade'
565     | 'update-gconf-defaults'
566     | 'register'
567     | 'unregister'
568     | 'OLD_DIR'
569     | 'SCHEMA_FILES'
570     | 'SCHEMA'
571     | 'purge'
572     | 'rm'
573     | 'rmdir'
574     | 'ignore-fail-on-non-empty'
575     | 'update-icon-cache'
576     | 'quiet'
577     | 'install-info'
578     | 'update-rc.d'
579     | 'ldconfig'
580     | 'update-menus'
581     | 'PYTHON'
582     | 'DIRLIST'
583     | 'scrollkeeper-update'
584     | 'update-mime-database'
585     | 'usr'
```

```
586     | 'share'
587     | 'mime'
588     | 'lib'
589     | 'compileall.py'
590     | 'emacsen-common'
591     | 'emacs-package-install'
592     | 'emacs-package-remove'
593     | 'etc'
594     | 'gconf'
595     | 'schemas'
596     | 'init.d'
597     | '_dh_action'
598     | 'restart'
599     | 'start'
600     | 'invoke-rc.d'
601     | 'stop'
602     | 'inst'
603     | 'menu-methods'
604     | 'a+x'
605     | 'a-x'
606     | 'catalog'
607     | 'super'
608     | 'update-catalog'
609     | 'old'
610     | 'suid.conf'
611     | 'suidunregister'
612     | 'update-alternatives'
613     | 'install'
614     | 'bin'
615     | 'x-window-manager'
616     | 'boot'
617     | 'System.map'
618     | 'depmod'
619     | 'add'
620     | 'update-fonts-dir'
621     | 'md5sum'
622     | 'sed'
623     | 's'
624     | 'p'
625     | 'dpkg-query'
626     | 'Conffiles'
627     | 'debconf'
628     | 'confmodule'
629     | 'db_purge'
630     | 'slave'
631     | 'man'
632     | 'man1'
633     | 'x-window-manager.1.gz'
634     | 'suidregister'
635     | 'chown'
636     | 'chmod'
637     | 'read'
638     | 'dir'
639     | 'true'
640     | 'echo'
641     | 'Preserving'
642     | 'user'
643     | 'changes'
644     | 'to'
645     | 'mv'
646     | 'line'
647     | 'mode'
648     | 'mkdir'
649     | 'group'
650     | 'dpkg'
651     | 'awk'
652     | 'py'
653     | 'print'
654     | 'c'
655     | 'xargs'
```

```
656     | 'n'
657     | 'set'
658     | '*'
659     | 'q'
660     | 'dev'
661     | 'null'
662     | 'i'
663     | 'doc-base'
664     | 'r'
665     | 'x'
666     | 'd'
667     | 'f'
668     | ID
669     ;
670
671 DIGIT   : '0'..'9';
672
673 ID  : ('a'..'z' | 'A'..'Z' | '*' | '+')( 'a' .. 'z' | 'A' .. 'Z' | '0'
674 .. '9' | UNDERSCORE | MINUS | QMARK | DOT | '+' | LSBRACK | RSBRACK)*;
675
676 AMPERSAND   : '&';
677
678 COLON   : ':';
679
680 VBAR    : '|';
681
682 DOLLAR  : '$';
683
684 UNDERSCORE : '_';
685
686 QMARK   : '?';
687
688 SHARP   : '#';
689
690 EXCL    : '!';
691
692 MINUS   : '-';
693
694 SLASH   : '/';
695
696 BSLASH  : '\\';
697
698 DMINUS  : '--';
699
700 AND : '&&';
701
702 OR  : '||';
703
704 LSBRACK : '[';
705
706 RSBRACK : ']';
707
708 LPAR    : '(';
709
710 RPAR    : ')';
711
712 LGRAF   : '{';
713
714 RGRAF   : '}';
715
716 DQUOTE  : '"';
717
718 DOT : '.';
719
720 AP          : '\u0060';
721
722 EQU : '=';
723
724 NOTEQU  : '!=';
725
```

```
726 DLESS␣␣␣:␣'<<';
727
728 DGREAT␣␣:␣'>>';
729
730 LESSAND␣:␣'<&';
731
732 GREATAND␣␣␣␣:␣'>&';
733
734 LESSGREAT␣␣␣:␣'<>';
735
736 DLESSDASH␣␣␣:␣'<<-';
737
738 CLOBBER␣:␣'>|';
739
740 LESS␣␣␣␣:␣'<';
741
742 GREAT␣␣␣:␣'>';
743
744 ANDGREAT␣␣␣␣:␣'&>';
745
746 DSEMI␣␣␣:␣';;';
747
748 APSINGLE␣␣␣␣:␣'\u0027';
749
750 TILDE␣␣␣:␣'\u007E';
751
752 COMMENT␣@init␣{
753 ␣␣␣␣skip();
754 }
755 ␣␣␣␣:␣{getCharPositionInLine()>0}?␣=>␣␣~(BSLASH)␣SHARP␣~(EXCL)␣(~('\n'␣|␣'\r'))*␣'\r'?
756 ␣␣␣␣|␣{getCharPositionInLine()==0}?␣=>␣(␣'␣'␣|␣'\t')*␣SHARP␣~(EXCL)␣(~('\n'␣|␣'\r'))*␣'\
    ↪r'?␣'\n'
757 ␣␣␣␣;
758
759 BLANKLINE
760 ␣␣␣␣:␣␣{getCharPositionInLine()==0}?␣=>␣(␣'␣'␣|␣'\t')*␣'\n'␣{skip();};
761
762 NOTNEWLINE␣␣:␣(BSLASH␣'\n')␣{skip();};
763
764 WS␣␣:␣(␣'␣'␣|␣'\t')+␣{␣$channel=HIDDEN;};
```

Bibliography

[AGRH05] Juan-José Amor-Iglesias, Jesús M. González-Barahona, Gregorio Robles-Martínez, and Israel Herráiz-Tabernero. Measuring Libre Software Using Debian 3.1 (Sarge) as A Case Study: Preliminary Results. *Upgrade Magazine*, August 2005.

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

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

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

[CDRP+10] Antonio Cicchetti, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. *A Model Driven Approach to Upgrade Package Based Software Systems*, chapter Communications in Computer and Information Science. Springer, 2010. In press.

[CIM09] Javier Luis Cánovas Izquierdo and Jesús García Molina. A domain specific language for extracting models in software modernization. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 82–97, Berlin, Heidelberg, 2009. Springer-Verlag.

[CRP+09] Antonio Cicchetti, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Towards a model driven approach to upgrade complex software systems. In *proceedings of the 4th International Working Conference on Evaluation of Novel approaches to Software Engineering (ENASE 2009)*, Milan - Italy, 6 - 10 May 2009.

[DCZT08] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in foss distributions: details and challenges. In *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, New York, NY, USA, 2008. ACM.

[DH07] Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *USENIX'07*, pages 1–6, San Diego, CA, 2007.

[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.

[DTP+09] Davide Di Ruscio, John Thomson, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. First version of the dsl based on the model developed in wp2.

Mancoosi Project deliverable D3.2, January 2009. http://www.mancoosi.org/reports/d3.2.pdf.

[Ecl] Eclipse. Modisco project. Available: http://www.eclipse.org/gmt/modisco/.

[EDO06] EDOS Project. Report on formal management of software dependencies. EDOS Project Deliverable D2.1 and D2.2, March 2006.

[Eff06] S. Efftinge. openarchitectureware 4.1 xtext language reference, August 2006. http://www.eclipse.org/gmt/oaw/doc/4.1/r80 xtextReference.pdf.

[Eil05] Eldad Eilam. *Reversing: Secrets of reverse engineering*. Wiley Publishing, Inc., 2005.

[FBB+07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *MoDELS*, pages 482–497, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[JBB09] Frédéric Jouault, Jean Bézivin, and Mikaël Barbero. Towards an advanced model-driven engineering toolbox. *ISSE*, 5(1):5–12, 2009.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of GPCE'06*, pages 249–254. ACM, 2006.

[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.

[JS08] Ian Jackson and Christian Schwarz. Debian policy manual. http://www.debian.org/doc/debian-policy/, 2008.

[KU07] V. Khusidman and W. Ulrich. Architecture-driven modernization: Transforming the enterprise. Technical report, Tactical Strategy Group: white paper, 2007.

[Mav08] Apache maven project. http://maven.apache.org/, 2008.

[Nie08] Gustavo Niemeyer. Smart package manager. http://labix.org/smart, 2008.

[Nor08] Gustavo Noronha Silva. APT howto. http://www.debian.org/doc/manuals/apt-howto/, 2008.

[Obj03] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04*. http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[RGvD06] Thijs Reus, Hans Geers, and Arie van Deursen. Harvesting software systems for mda-based reengineering. In *ECMDA-FA*, pages 213–225, 2006.

[RPPZ09] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Towards Maintainer Script Modernization in FOSS Distributions. In *proceedings of the IWOCE2009 - Open Component Ecosystems International Workshop - colocated with ESEC/FSE 2009*, Amsterdam, The Netherlands, 24 August 2009.

[Sch06] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[sPL03] Robert C. seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[WK06] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of LNCS, pages 159–168. Springer-Verlag, 2006.