mancoosi

managing software complexity

Metamodel for Describing System Structure and State

Deliverable 2.1

Nature : Deliverable
Due date : 31.01.2009
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> Stefano Zacchiroli <zack@pps.jussieu.fr>
Internal review	Sophie Cousin, Arnaud Laprevote, Paulo Trezentos
Workpackage number	WP2
Deliverable number	1
Document type	Deliverable
Version	1
Due date	31/01/2009
Actual submission date	03/02/2009
Distribution	Public
Project coordinator	Roberto Di Cosmo <roberto@dicosmo.org>

Abstract

Today's software systems are very complex modular entities, made up of many interacting components that must be deployed and coexist in the same context. Modern operating systems provide the basic infrastructure for deploying and handling all the components that are used as the basic blocks for building more complex systems even though a generic and comprehensive support is far from being provided. In fact, in Free and Open Source Software (FOSS) systems, components evolve independently from each other and because of the huge amount of available components and their different project origins, it is not easy to manage the life cycle of a distribution. Users are in fact allowed to choose and install a wide variety of alternatives whose consistency cannot be checked a priori to their full extent. It is possible to easily make the system unusable by installing or removing some packages that "break" the consistency of what is installed in the system itself.

This document proposes a model-driven approach to simulate system upgrades in advance and to detect predictable upgrade failures, possibly by notifying the user before the system is affected. The approach relies on an abstract representation of the systems and packages which are given in terms of models that are expressive enough to isolate *inconsistent configurations* (e.g., situations in which installed components rely on the presence of disappeared sub-components) that are currently not expressible as inter-package relationships.

Contents

List of Figures

Chapter 1

Introduction

Traditionally, software systems are designed with a set of requirements and pre-established context assumptions. On the contrary, the modern society asks for software systems for which the evolution of both requirements and context is the norm rather than the exception. A clear, yet extreme, example is offered by network services, where the context frequently changes to the points where systems *must* be changed in the shortest possible time frame to account for security software upgrades. When the requirements and the context change that rapidly, evolving software system must be able to face changes in an effective way with minimal human intervention otherwise it will soon become obsolete. Mechanisms for run-time evolution are needed to manage system and context changes. The joint ability to effectively react to changes without degrading the level of dependability is the key factor for delivering successful systems that continuously satisfy evolving user requirements [SS04].

In the domain of Component-Based ("CB" for short) software systems [Szy98, Szy03], run-time evolution deals with the difficult problems arising when one wants to efficiently and safely modify the set of the software components and their interactions in complex software infrastructures. In fact, a CB system is an assembly of software components (usually implemented by means of either third-party libraries or in-house components), designed to meet the system requirements that were identified during the analysis phase. In this setting, the addition of new components, the deletion or upgrade of existing ones need to be effectively managed in order not to compromise the correct run-time behavior of the overall system [BTLd].

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. These usually contain *maintainer scripts*, which are executed during the upgrade process to finalize component configuration. The adopted scripting languages have rarely been formally investigated, thus posing additional difficulties in understanding their side-effects which can spread throughout the system. In other words, even though a package might be viewed as a software unit, it lives without a proper component model which usually defines standards (e.g., how a component interface has to be specified and how components communicate) [Szy98, Szy03] that facilitate integration assuring that components can be upgraded in isolation.

The problem of maintaining FOSS installations, or similarly structured software distributions, is intrinsically difficult and is missing a satisfactory solution. Today's available package managers lack several important features such as complete dependency resolution and roll-back of failed upgrades [DTZ08]. Moreover, there is no support to simulate upgrades taking the behavior of maintainer scripts into account. In fact, current tools mostly consider only inter-package relationships which are not sufficient to predict side-effects and system inconsistencies which are encountered during upgrades. It is however important to take into account maintainer scripts since they can have system-wide effects, and hence cannot be narrowed to the involved packages only. In this respect, proposals like [Oli04, TDL+07] represent a first step toward roll-back management. In fact, they support the re-creation of removed packages on-the-fly, so that they can be re-installed to undo an upgrade. However, such approaches can track only files which are under package manager control and, in some cases, only files flagged as "configuration files". Therefore, unlike us, none of such approaches can undo maintainer script side effects.

An interesting proposal to support the upgrade of a system, called NixOS, is presented in [DL08]. It is a purely functional distribution meaning that all static parts of a system (such as software packages, configuration files and system startup scripts) are built by pure functions. Among the main limitations of NixOS there is the fact that some actions related to upgrade deployment can not be made purely functional (e.g., user database management). Moreover, NixOS solution to the upgrade problem does not consider maintainer scripts.

[McQ05] proposes an attempt to monitor the upgrade process with the aim to discover what is actually being touched by an upgrade. Unfortunately, it is not sufficient to know which files have been involved in the maintainer scripts execution but we also have to consider system configuration, running services etc., as taken into account by our metamodels. Even focusing on touched files only, it is not always possible to undo an upgrade by simply recopying the old file [DTZ08].

Interesting techniques for static analysis of (shell) scripts have also been proposed. Some previous work [XA06] deals with SQL injection detection for PHP scripts, but it did not consider the most dynamic parts of the PHP language, quite common in scripting languages. Whereas, [MZ07] presents an "arity" bug detection in shell scripts, but once more only considers a tiny fragment of the shell language. Both works are hence far even from the minimal requirement of determining a priori the set of files touched by script execution, letting aside how restricted were the considered shell language subsets. Given these premises, we are skeptical that static analysis can fully solve the problem illustrated in our work.

On the contrary, we propose a model-driven approach to specify system configurations and available packages. Even maintainer scripts are described in terms of models which abstract from the real system, but are expressive enough to predict several of their effects on package upgrades. This is an innovative aspect which distinguishes our proposal from the existing systems summarized above. Intuitively, we provide an abstract interpretation of scripts, in the spirit of [Cou06], which focuses on the relevant aspects to predict the operation effects on the software distribution. To this end, models can be used to drive roll-back operations to recover previous configurations according to user decisions or after upgrade failures. To summarize, Mancoosi works on models to (i) simulate the execution of maintainer scripts, (ii) predict side-effects and system inconsistencies which might be raised by package upgrades, and (iii) instruct roll-back operations to recover previous configurations according to user decisions or after upgrade failures.

File modifications performed by users between transactions are not supported by model-based rollback, since they cannot be captured in the model. However, versioning systems might be

taken into account in conjunction with the proposed model-based approach in order to maintain the different versions of the files which have been manually modified.

1.1 Structure of the deliverable

The present deliverable is structured as follows:

- Chapter 2 describes the life-cycle of FOSS distributions with particular attention to the upgrade process of FOSS packages;
- Chapter 3 describes the use of models for supporting the upgradeability of FOSS distributions. After an introduction to Model Driven Engineering (MDE), we present the Mancoosi model driven approach to (i) specify system configurations and packages, (ii) simulate the installation of software packages and (iii) assist roll-backs;
- Chapter 4 analyzes the FOSS domain, in particular the Debian and RPM-based distributions, stemming out the required modeling elements. The analysis has been performed in several steps both automated and manual and reached a satisfactory coverage of the considered scripts;
- Building on that, Chapter 5 presents the metamodels on which the Mancoosi approach is based. In particular, three interrelated metamodels are proposed: (i) a package metamodel, which describes the metamodel of packages that compose the system (ii) a configuration metamodel for modeling package settings, and (iii) a log metamodel to store the operations performed in upgrading and to instruct the rollback. For this reason it is based on the concept of transactions which represent a set of statements which change the system configurations;
- Chapter 6 presents techniques to support the evolution of the proposed metamodels and the automatic adaptation of the already existing models which conform to the changed metamodels;
- Finally, Chapter 7 concludes the deliverable by summarizing its content.

1.2 Glossary

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

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

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

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

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

Model According to Mellor et al. [MCF03] a model "is a coherent set of formal elements describing something (e.g., a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis" such as communication of ideas between people and machines, test case generation, transformation into an implementation etc. Moreover, a model is defined to answer questions in place of the actual system.

Metamodel A metamodel consists of "concepts" (things, terms, etc.) proper of a certain domain. It's an abstraction which highlights properties of models which are said to *conform to* its *metamodel* like a program conforms to the grammar of the programming language in which it is written [Bó5].

Model transformation Kleppe et al. [KW03] defines a *model transformation* as the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed to a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed to one or more constructs in the target language.

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

Package status A set of metadata maintained by the installer about packages currently installed on a machine. The package status is used by the installer as a model of the software installed on a machine and kept up to date upon package installation and removal. The kind of metadata stored for each package varies from distribution to distribution, but typically comprises package identifiers (usually name and version), human-oriented information such as a description of what the package contains and a formal declaration of the inter-package relationships of a package. Inter-package relationships can usually state

package requirements (which packages are needed for a given one to work properly) and conflicts (which packages cannot coexist with a given one).

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

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

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

Chapter 2

Standard life-cycle of FOSS distributions

Overall, the architectures of all FOSS distributions are similar. Each user machine has a local *package status* recording which packages are currently installed and which are available from remote repositories. Package managers are used to manipulate the package status and can be classified in two categories [EDO06]: *installers*, which deploy individual packages on the filesystem (possibly aborting the operation if problems are encountered) and *meta-installers*, which act at the inter-package level, solving dependencies and conflicts, and retrieving packages from remote repositories as needed.

We use the term *upgrade problem* to refer generically to any request (e.g., install, remove, upgrade to a newer version) that change the package status. Such problems are usually solved by meta-installers, the aim of which is to find a suitable *upgrade plan*, if one exists. In the rest of the chapter we give a brief description of packages (as they can be found in current distributions), their role in the upgrade process, and the failures that can impact on upgrade deployment.

2.1 Packages

Abstracting over format-specific details[1], a *package* is a bundle of three main parts:

Package {
- 1. Set of *files*
  - 1.1. *Configuration files*
- 2. Set of valued *meta-information*
  - 2.1. *Inter-package relationships*
- 3. Executable *configuration scripts*

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

Configuration files (1.1) is a distinguished subset of shipped files, identifying those affecting the runtime behavior of the package and meant to be locally customized with or without package

[1] Such as those of the `.deb` package format found in distributions derived from Debian (`http://www.debian.org`), or of the `.rpm` format found in distributions derived from Red Hat (`http://www.redhat.com`).

manager mediation. Configuration files need to be present in the bundle (e.g., to provide sane defaults or documentation), but need special treatment: during installation of new versions of a package, they cannot be simply overwritten, as they may contain local changes.

Package meta-information (2) contains information which varies from distribution to distribution. A common core provides: a unique identifier, software version, maintainer and package description, but most notably, distributions use meta-information to declare *inter-package relationships* (2.1). The relationship kinds vary with the installer, but there exists a de facto common subset including: dependencies (the need of other packages to work properly), conflicts (the inability of being co-installed with other packages), feature provisions (the ability to declare named features as provided by a given package, so that other packages can depend on them), and restricted boolean combinations of them [EDO06].

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

Example 2.1 *A maintainer script embodying one of the most common use case (see Chapter 4) is the following shell script snippet, to be executed in the* postinst *phase (i.e., after having copied package files on disk) of a shared library package:*

```
1 if [ "$1" = "configure" ]; then
2     ldconfig
3 fi
```

This script simply invokes an external program to update the Linux run-time linker cache.

The following facets of maintainer scripts are noteworthy:

1. Maintainer scripts are full-fledged programs, written in Turing-complete programming languages. They can do anything permitted to the installer, which is usually run with system administrator rights;
2. The functionality of maintainer scripts cannot be substituted by just shipping extra files: the scripts often rely on data which is available only in the target installation machine, and not in the package itself;
3. Maintainer scripts are required to work "properly": upgrade runs, in which they fail, trigger upgrade failures and are usually detected via inspection of script exit code;
4. Maintainer scripts should not require any user interaction or any other external potential blocking calls such as HTTP request to outside servers and CD-ROM access. Given that in some scenarios (e.g. distribution upgrade, automatic security updates, etc.) packages are upgraded in batch this would make it unmanageable.

2.2 Upgrades

Table 2.1 summarizes the different phases of what we call the *upgrade process*, using as an example the popular `APT` meta-installer [Nor08]. The process starts in phase (1) with the

`# apt-get install libapache2-mod-php5`	(1) request

`Reading package lists... Done`
`Building dependency tree... Done`
 	
`The following NEW packages will be installed:`
`  libapache2-mod-php5`
`0 upgraded, 1 newly installed, 0 to remove and`
`  0 not upgraded.`
`Need to get 2543kB  of archives. After this`
`operation, 5743kB of additional disk space`
`will be used.`
	(2) dep. resolution

`Get:1 http://va.archive.ubuntu.com`
`hardy-updates/main libapache2-mod-php5`
`5.2.4-2ubuntu5.3 [2543kB]`
`Fetched 2543kB in 2s(999kB)`
	(3) package retrieval

	(5a) pre-configuration

`Selecting package libapache2-mod-php5.`
`(Reading database ... 162440  files and`
`dirs installed.)`
`Unpacking libapache2-mod-php5`
`(from .../libapache2-mod-php5_5.2.4-2`
`ubuntu5.3_i386.deb)`
	(4) unpacking

`Setting up libapache2-mod-php5` `(5.2.4-2ubuntu5.3)`	(5b) post-configuration

Table 2.1: The package upgrade process

user requesting to alter the local package status. The expressiveness of the requests varies with the meta-installer, but the aforementioned actions (install, remove, etc.) are ubiquitously supported.

Once the user request is known, an *upgrade problem* is properly defined. Abstractly, we can define it as a triple $\langle U, S_o, R\rangle$, where U is a distribution (i.e., a set of packages), $S_o \subseteq U$ is a package status, and R a user request; its *solutions* are all possible package status $S \subseteq U$, such that:[2]

a. The user request R is satisfied by S;

b. If S contains a package p, it contains all its dependencies;

c. S does not contain two conflicting packages;

d. S has been obtained executing all required hooks and none of the involved maintainer scripts has failed.

Phase (2) checks whether a package status satisfying (b) and (c) above exists (the problem is at least NP-complete [EDO06]). If this is the case one is chosen in this phase. Deploying the new status consists of package retrieval, phase (3), and unpacking, phase (4). Unpacking is the first phase actually changing both the package status (to keep track of installed packages) and the filesystem (to add or remove the involved files).

[2] A few remarks: while (a) is installer-specific, (b) and (c) have been generalized and formalized in [MBC+06]; studies of (d) are still lacking. These are just the *functional* properties of an upgrade outcome, but there are also *non-functional* properties that can be used to choose *optimal* solutions (e.g., minimality of change, or downtime length); this issue is among the objectives of workpackages WP4 and WP5. Note that while checks for (b) and (c) can be performed statically, checks for (d) can only be performed at run-time while executing scripts.

During unpacking, configuration files are treated by checking whether local configuration files have been manually modified or not. In this case *merging* is required and it is typically done by asking the user to manually do it.

Intertwined with package retrieval and unpacking, there can be several configuration phases, (exemplified by phases (5a) and (5b) in Table 2.1), where maintainer scripts get executed. The details depend on the available hooks.

Example 2.2 *The installation of PHP5 (a web scripting language integrated with the Apache web server) executes the following code defined in a* postint *script:*

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

The Apache module `php5`*, installed during the unpacking phase, gets enabled by the above snippet invoking the* `a2enmod` *command in line 3; the Apache service is then reloaded (line 4) to make the change effective. Upon PHP5 removal the reverse will happen, as implemented by PHP5* prerm *script:*

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

Note that prerm *is executed* before *removing files from disk, that is necessary to avoid reaching an inconsistent configuration where the Apache server is configured to rely on no longer existing files. It is important to observe that the expressiveness of inter-package dependencies is not enough to encode this kind of dependencies: Apache does not depend on* `php5` *(and should not, because it is also useful without it), but when* `php5` *is installed, Apache needs specific configuration to work in harmony with it. The bookkeeping of such configuration is delegated to maintainer scripts.*

2.3 Failures

Each phase of the upgrade process can fail. Dependency resolution can fail either because the user request is unsatisfiable (e.g., user error or inconsistent distributions [MBC+06]) or because the meta-installer is unable to find a solution. Completeness, i.e., the guarantee that a solution will be found whenever one exists, is a desirable meta-installer property unfortunately missing in most meta-installers, with too few claimed exceptions [TSJL07].

While (ad-hoc) SAT solving has proved to be a suitable complete technique to solve dependencies [MBC+06], *handling of user preferences* is a novel problem for package upgrades. It boils down to let users specify which solution to choose among all acceptable solutions. Examples of preferences are not only policies [Nie08] like minimizing the download size or prioritizing popular packages, but also more customized requirements such as blacklisting packages maintained by an untrusted maintainer.

Package deployment can fail as well. Trivial failures, e.g., network or disk failures, can be easily dealt with when considered in isolation from the other upgrade phases: the whole upgrade process can be aborted and unpack can be undone, since all the involved files are known. Maintainer script failures cannot be as easily undone or prevented, given that all non-trivial

properties about scripts are undecidable, including determining *a priori* which parts of file-system they affect to revert them *a posteriori*.

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

To the ends of system modeling, undetected failures are relevant since they show the need of "out of order" undo. For instance, if three upgrades u_1, u_2, and u_3 are performed subsequently and if u_1 induced an undetected failure, it is not desirable to require the undo of all three upgrades to counter the failure. It is rather desirable to be able to selectively undo only u_1, because several time has possibly elapsed since u_1 and users might already have started to rely on the (positive) effects of u_2 and u_3.

Chapter 3

Models for supporting the upgrades in FOSS distributions

The problem of maintaining FOSS installations is far from trivial and has not been properly addressed yet [DTZ08]. In particular, current package managers are neither able to *predict* nor to *counter* vast classes of upgrade failures. The reason is that package managers rely on package meta-information only (in particular on inter-package relationships), which are not expressive enough. Our proposal consists in maintaining a model-based description of the system and simulate upgrades in advance on top of it, to detect predictable upgrade failures and notify the user before the system is affected. More generally, the models are expressive enough to isolate *inconsistent configurations* (e.g., situations in which installed components rely on the presence of disappeared sub-components), which are currently not expressible as inter-package relationships.

In this chapter we promote the adoption of model-driven techniques since they present several advantages: *a)* models can be given at any level of abstraction depending on the analysis and operations one would like to perform as opposed to actual package dependency information whose granularity is fixed and often too coarse; *b)* complex and powerful analysis techniques are already available to detect model conflicts and inconsistencies [MSD06, CRP08]. In particular, contradictory patterns can be specified in a structural way by referring to the domain underlying semantics in contrast with text-based tools like version control systems where conflicts are defined at a much lower level of abstraction as diverging modifications of the same lexical element.

The remaining of the chapter is organized as follows: Section 3.1 provides the reader with the essential concepts of MDE which underpin the approach supporting the upgradeability process which is described in Section 3.2.

3.1 Model Driven Engineering

Model-Driven Engineering (MDE) [Sch06b] refers to the systematic use of models as first class entities throughout the software engineering life cycle. Model-driven approaches shift development focus from third generation programming language codes to models expressed in proper domain specific modeling languages. The objective is to increase productivity and reduce time to market by enabling the development of complex systems by means of models defined with

concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain. This makes the models easier to specify, understand, and maintain [Sel03] helping the understanding of complex problems and their potential solutions through abstractions.

The concept of Model Driven Engineering emerged as a generalization of the Model Driven Architecture (MDA) proposed by the Object Management Group (OMG) in 2001 [Obj03a] and it relies on a conceptual framework consisting of *model*, *meta-model*, and *model transformation* which are described in the rest of the section.

3.1.1 Models and Meta-models

Even though MDA and MDE rely on *models* that are considered "first class citizens", there is no common agreement about what a model is. In [Sei03] a model is defined as "a set of statements about a system under study". Bézivin and Gerbé in [BG01] define a model as "a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system". According to Mellor et al. [MCF03] a model "is a coherent set of formal elements describing something (e.g., a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis" such as communication of ideas between people and machines, test case generation, transformation into an implementation, etc. The MDA guide [Obj03a] defines a model of a system as "a description or specification of that system and its environment for some certain purposes. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language". These formulations do not conflict but rather complement one another and represent the various aspects of the fundamental philosophical category of software. Models are generally used in order to pursue the following purposes (amongst others):

- *to record design decisions*, models are generally existing prior to software and are used either to convey relevant information to the implementors or to be manipulated in tool chains; the outcome consists in both cases of artefacts, which can range from documentation and component interfaces to software skeletons or full-fledged systems. The process typically enriches subsequent models with details which derive from the knowledge the manipulations rely on, regardless whether automated or manual;

- *to analyze a system*, highly abstract and incomplete models are verified possibly using a computer to analyze the system for the presence of desirable properties and the absence of undesirable ones. This can be done in several ways, including formal (mathematical) analyses such as performance analysis based on queuing theory or safety-and-liveness property checking. The analyses can be performed before or after the system comes into existence depending on the needs.

Techniques and tools to support model-driven engineering nowadays reached a certain degree of maturity making it practical even in large-scale industrial applications. For a detailed discussion about the use and meaning of models in MDE please refer to [Bó5].

In MDE models are not considered as merely documentation but precise artifacts that can be understood by computers and can be automatically manipulated. In this scenario *meta-modeling* plays a key role. It is intended as a common technique for defining the abstract syntax of models and the interrelationships between model elements. Meta-modeling can be seen as the construction of a collection of "concepts" (things, terms, etc.) within a certain

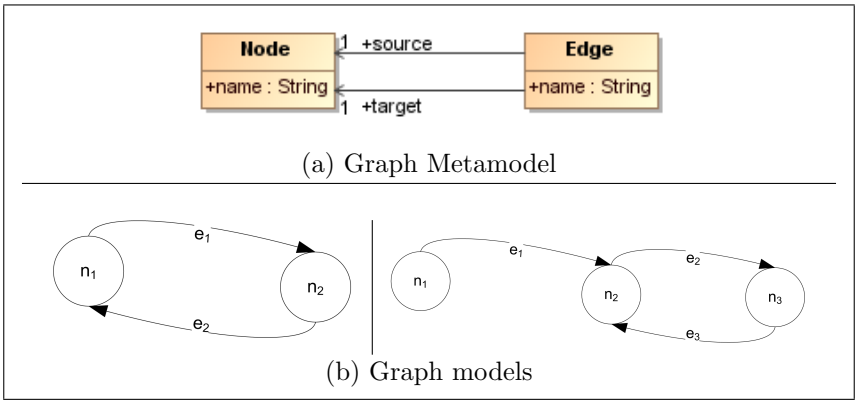

Figure 3.1: Models conforming to a sample metamodel

domain. A model is an abstraction of phenomena in the real world, and a meta-model is yet another abstraction, highlighting properties of the model itself. This model is said to *conform to* its *meta-model* like a program conforms to the grammar of the programming language in which it is written [Bó5]. For instance, Fig. 3.1.a depicts a sample metamodel containing the concepts and the relations proper of graphs. In this respect, the metamodel contains the concept `Node` which represents a source and/or target of edges according to the relations between the `Node` and `Edge` metaclasses. In Figure 3.1.b two sample models conforming to graph metamodels previous mentioned are reported.

OMG has introduced the four-level architecture illustrated in Fig. 3.2. At the bottom level, the `M0` layer is the real system. A model represents this system at level `M1`. This model conforms to its meta-model defined at level `M2` and the meta-model itself conforms to the metametamodel at level `M3`. The metametamodel conforms to itself. OMG has proposed Meta Object Facility (MOF) [Obj03b] as a standard for specifying meta-models. For example, the Unified Modeling Language (UML) meta-model [Obj03c] is defined in terms of MOF. A supporting standard of MOF is XML Metadata Interchange (XMI) [Obj03d], which defines an XML-based exchange format for models on the `M3`, `M2`, or `M1` layer. This metamodeling architecture is common to other technological spaces as discussed by Kurtev et al. in [AKB02]. For example, the organization of programming languages and the relationships between XML documents and XML schemas follow the same principles described above (see Fig. 3.2). In addition to metamodeling, *model transformation* is also a central operation in MDE as discussed in the next section.

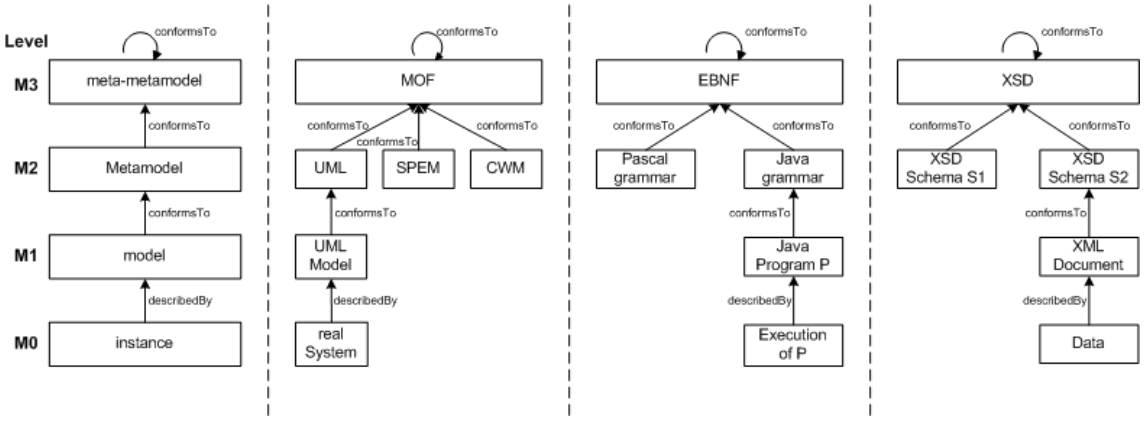

Figure 3.2: The four layers meta-modeling architecture

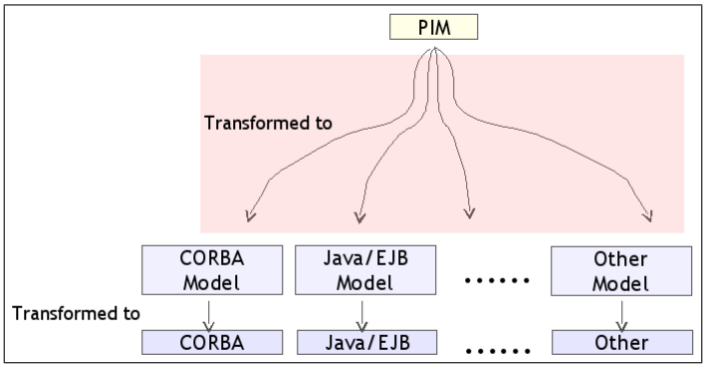

Figure 3.3: MDA based development Process

3.1.2 Model Transformations

In addition to metamodeling, *model transformation* is also a central operation in MDA as depicted in Fig. 3.3. According to the figure, the development of a software system starts by building a *platform independent models* (PIM) of that system. Then the PIM is refined and transformed to one or more *platform specific models* (PSMs). Finally, the PSMs are transformed to code. In this way, MDA allows us to preserve the investments in business logic since, being a PIM totally unrelated to any specific technology, it is possible to map it to different platforms by means of (semi)automatic transformations which can be defined according to specific needs. Achieving this goal would enable analysts to focus only on the design, the business logic, and the overarching architecture.

The MDA guide [Obj03a] defines a model transformation as "the process of converting one model to another model of the same system". Kleppe et al. [KW03] defines a *transformation* as the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed to a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed to one or more constructs in the target language.

Rephrasing these definitions by considering Fig. 3.4, a model transformation program takes as input models conforming to a given source meta-model and produces as output other models conforming to a target meta-model. The transformation program, composed of a set of rules, should itself considered as a model. As a consequence, it is based on a corresponding meta-model, that is an abstract definition of the used transformation language. Generating lower-level models, and eventually code, from higher-level models is not the unique use of model transformations which can be adopted also to support:

- *the synchronization of models at the same level or different levels of abstraction*, the specification of complex and large systems might consist of several models which have to be kept synchronized. This means that a modification performed on a given model has to be propagated to the other ones.

- *the creation of query-based views on a system*, complex models can be queried to extract simpler views which are more suitable for automated manipulation and analysis;

- *the co-evolution of models*, similarly to other software artefacts, metamodels can evolve over time too. Accordingly, models need to be co-adapted in order to remain compliant

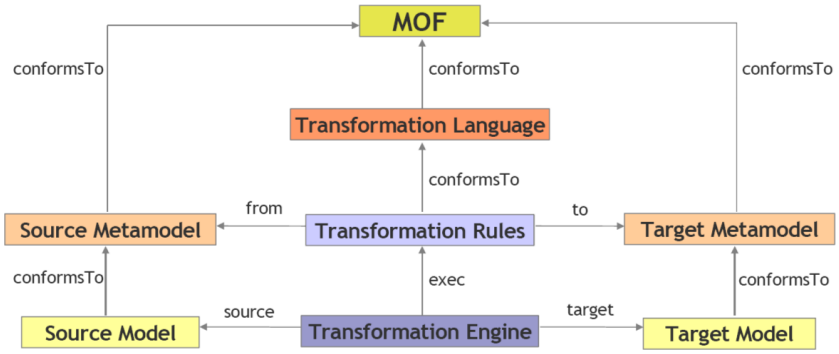

Figure 3.4: Basic Concepts of Model Transformation

to the metamodel and not become eventually invalid. Model transformations can be used to adapt existing models with respect to the metamodel modifications;

- *the reverse engineering of higher-level models from lower-level ones*, specific model transformations can be developed to harvest existing systems and generate corresponding models which are more abstract and amenable to analyze the system with respect to desirable properties.

These aspects will be discussed with more details in Chapter 6, which suggests an approach to support the evolution of the MANCOOSI metamodels presented in Chapter 5.

Many languages and tools have been proposed to specify and execute transformation programs. In 2002 OMG issued the Query/View/Transformation request for proposal [Obj02] to define a standard transformation language. Even though a final specification has been adopted at the end of 2005, the area of model transformation continues to be a subject of intense research. Over the last years, in parallel to the OMG process a number of model transformation approaches has been proposed both from academia and industry. The paradigms, constructs, modeling approaches and tool support distinguish the proposals, each of them with a certain suitability for a certain set of problems.

3.2 MDE and FOSS distributions upgrades

In this section we propose a model-driven approach to support the upgradeability of FOSS distributions built of composable units which evolve independently. The approach relies on the specification of system configurations and available packages. Maintainer scripts are also described in terms of models which abstract from the real system, but are expressive enough to predict several of their effects on package upgrades. Intuitively, we provide a more abstract interpretation of scripts, in the spirit of [Cou06], which focuses on the relevant aspects to predict the operation effects on the software distribution. To this end, models can be used to drive rollback operations to recover previous configurations according to user decisions or after upgrade failures.

To simulate an upgrade run, two models are taken into account (see Fig. 6.8): the *System Model* and the *Package Model* (see the arrow ⓐ). The former describes the state of a given system in terms of installed packages, running services, configuration files, etc. The latter provides information about the packages involved in the upgrade, in terms of inter-package relationships. Moreover, since a trustworthy simulation has to consider the behavior of the maintainer scripts

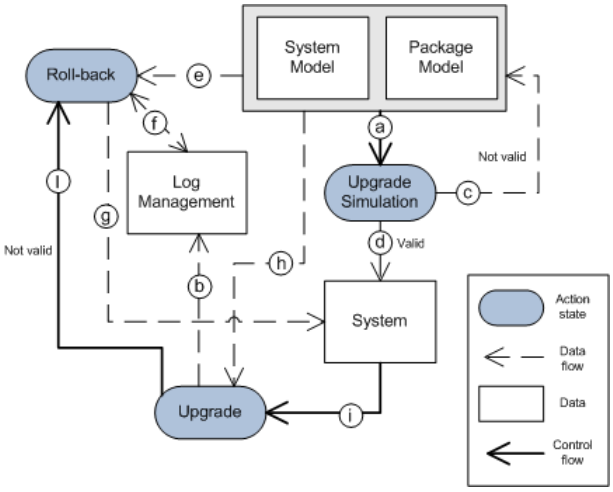

Figure 3.5: Proposed approach

which are executed during the package upgrades, the package model specifies also an abstraction of the behaviors of such scripts. There are two possible simulation outcomes: *not valid* and *valid* (see the arrows ⓒ and ⓓ, respectively). In the former case it is granted that the upgrade on the real system will fail. Thus, before proceeding with it the problem spotted by the simulation should be fixed. In the latter case—*valid*—the upgrade on the real system can be operated (see the arrow ⓘ). However, since the models are an abstraction of the reality, upgrade failures might occur due to reasons like a drive IO error.

During package upgrades *Log models* are produced to store all the transitions between configurations (see arrow ⓑ). The information contained in the system, package, and log models (arrows ⓔ and ⓕ) are used in case of failures (arrow ⓛ) when the performed changes have to be undone to bring the system back to the previous valid configuration (arrow ⓖ). Since it is not possible to specify in detail every single part of systems and packages, trade-offs between model completeness and usefulness have been evaluated. This analysis is reported in the next chapter. The result of such a study has been formalized in terms of metamodels (see Chapter 5) which can be considered one of the constituting concepts of Model Driven Engineering (MDE) [Sch06a]. They are the formal definition of well-formed models, constituting the languages by which a given reality can be described in some abstract sense [B́05] defining an abstract interpretation of the system.

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

All the models which are involved in the simulation upgrade are specified by using the modeling constructs formalized in specific metamodels which have been conceived during a domain analysis phase. In this respect, FOSS distributions (Debian installations with more attention) have been analysed as discussed in the next chapter. Subsequently, the elicited concepts are formalized in specific metamodels which are reported in chapter 5.

Chapter 4

Analysis of FOSS distributions

The first step that needs to be performed when defining a metamodel is to accurately study the domain in order to understand the elements and the artifacts that need to be modeled. This study helps to understand the right abstraction level, according to the use of the models that we will have. Therefore, we studied FOSS distributions and in particular the maintainer scripts that are the most critical part for our modeling needs. Maintainer scripts are executed at various points during the upgrade process to finalize component configuration. The adopted scripting languages are mainly POSIX shell. For instance, in the Debian Lenny distribution on a universe of 25'440 maintainer scripts, only 131 are written in Perl [PER09], 481 are given in Bash [BAS09] and the remaining 24'822 are implemented in POSIX-compliant Bourne shell [Bou].

Scripting languages have rarely been formally investigated and with no exciting results [XA06, MZ07], thus posing additional difficulties in understanding their side-effects which can spread throughout the whole system. Our aim is to describe maintainer scripts in terms of models which abstract from the real system, but are expressive enough to predict several of their effects on package upgrades. 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 analysis phase is then extremely important in order to find the right trade-off among expressiveness and abstraction.

Due to the large amount of scripts, we tried to collect them in clusters to be able to concentrate the analysis on representative of the equivalence classes identified.

The procedure we followed for clustering is as follows:

1. *Identify scripts generated from helpers:* a large number of scripts or part of them is generated by means of "helper" tools that provide a collection of small, simple and easily understood tools that are used to automate various common aspects of building a package. Since these (part of) scripts are automatically generated, for these scripts we can concentrate the analysis on the helpers themselves, rather than on the result of their usage.

2. *Identify scripts entirely composed of blank lines, comments, etc.*: these scripts are not interesting since they do not contain interesting statements.

   Nevertheless, after having analyzed part of the scripts which are generated from helper tools, it is possible that the remaining part of a given script is entirely composed of comments and blank lines. In those cases, the resulting "empty" scripts have been ignored, as their effects can be fully described in terms of their composing snippets coming from

helper tools.

3. *Study of scripts written "by hand"*: the remaining scripts need to be more carefully studied, as they have been written from scratch by package maintainers to address a specific need, most likely not covered by available helper tools. Actually we worked on identifying recurrent templates that maintainers use when writing the scripts.

The remaining of this chapter is organized as follows: Section 4.1 reports the analysis we performed on Debian, Section 4.2 shows the analysis we performed on RPM-based distribution, and Sections 4.3 and 4.4 conclude the chapter by stemming out elements that must be taken into account when defining the metamodels and elements that we are unable to cover, respectively.

4.1 Maintainer script analysis: Debian GNU/Linux

We have chosen the Debian GNU/Linux distribution[1] as the main data source for our analysis. The choice is motivated by the size—Debian is the largest package-based FOSS distribution in terms of packaged software—and by the representativeness of Debian—Debian is amongst the oldest distributions and possibly the one from which most derivative distributions have spun off. Morover, Debian packages rely substantially on maintainer scripts to perform sophisticated configuration handling.

The analysis has been performed considering a "snapshot" of Debian *Lenny*, the soon to be released "stable" brand of Debian. The snapshot has been taken on December 4th, 2008, considering only the `amd64` architecture (which will become the most widespread architecture on end-user machines in the near future), and all the packages shipped by the Debian archive and targeted at the end user (i.e., sections `main`, `contrib`, and `non-free`). Some aggregated figures about the distribution entailed by the considered snapshot are reported in the table below:

binary packages	22'823
source packages	12'681
size (binary only)	20.7 Gb
size (binary + source)	40 Gb

Each (binary) package[2] in Debian can come with 5 different kinds of maintainer scripts:

`preinst` (mnemonic for "pre-installation") script which is run before the files shipped by a package being installed have been unpacked on the filesystem of the target machine (see Section 2.2 for more details about the upgrade process and its unpacking phase).

`preinst` scripts are seldom used. A typical use case is to move away files belonging to other packages ("diverting" in Debian terminology) when they can get in the way of the package being installed.

`postinst` (mnemonic for "post-installation") script which is run after the files shipped by a package have been unpacked on the target filesystem.

[1] Debian GNU/Linux distribution website: `http://www.debian.org`.

[2] From now on, unless otherwise stated, we will use the term "package" to refer to binary packages, since that is the kind of packages users are faced with, and that defines the granularity at which software components can be installed or removed on a machine.

`postinst` scripts are possibly the most common maintainer configuration scripts and are typically used to update caches or other system-wide registries of information which (also) depend on files shipped by the just installed package.

`prerm` (mnemonic for "pre-removal script") which is the dual to `preinst`, since `prerm` scripts are executed just before removing from the target filesystem those files which belong to the package which is being removed.

`prerm` scripts are typically used to undo configuration actions which are strictly related to a package being removed, and which cannot be undone after the files composing that package have vanished from the filesystem (e.g., because they need a specific tool, which is part of the package being removed).

`postrm` (mnemonic for "post-removal") which is the dual to `postinst`, since `postrm` scripts are executed just after removing the files belonging to the package being removed from the filesystem.

`postrm` scripts are very often used to perform registry update actions similar to those performed by `postinst` scripts, because the updates need to be performed both just after having added and removed files which can possibly affect the registry content.

`config` (mnemonic for "configuration") script which is used to configure a software which requires specific user input to be configured. In particular, `config` scripts are usually paired with the `debconf`[3] configuration management system, which helps package maintainers to create interactions with the system administrator in order to ask for configuration parameters (e.g., "on which port you want to run the network service shipped by the just installed package"?).

Considering 5 maintainer scripts per package we obtain a (potential) universe of scripts to be considered of 114'115 scripts (i.e., $22'823 \times 5$). Luckily, 88'675 (77.7%) of those are actually missing from the corresponding packages; this means that those potential hooks are vacuously invoked during upgrades and do not need to be considered. The universe of the remaining scripts consists of "just" 25'440 scripts (22.3%).

4.1.1 Scripts generated from helpers

Package maintainers use complex toolchains to facilitate their maintenance work which is otherwise prone to repetition of self-similar tasks. In the specific case of Debian, the legacy helpers used, among other things, to generate (part of) maintainer scripts is the so called `debhelper` collection[4]. For the most part, `debhelper` consists of tools which are invoked at package build time to automate package-construction tasks such as installing specific file categories (e.g., manual pages, documentation, ...) in the location prescribed by the Debian policy [JS08].

Some of these tasks require the set up of configuration actions which are encoded as (part of) maintainer scripts.

Example 4.1 *All shared libraries which get installed (or removed) on a GNU/Linux machine typically requires triggering an update of the cache used by the Linux run-time linker, to speed up the loading of shared libraries. Once the `.so` files composing a shared library are in place on*

[3] `http://packages.debian.org/lenny/debconf`
[4] `http://packages.debian.org/lenny/debhelper`

the target filesystem, the cache can be updated by simply running the following command as the administrator:

```
ldconfig
```

From a packaging point of view, the aforementioned requirement is typically implemented by adding a shell script snippet both to the `postinst` *and to the* `postrm` *maintainer scripts. That way the linker cache is updated both just after the* `.so` *get installed upon package installation and just after they get removed upon package removal.*

Instead of requiring each maintainer to write exactly the same shell script snippets (which are needed by all packages shipping shared libraries) by hand, `debhelper` offers a template mechanism called "autoscripts" which writes down the needed snippets when needed.

Example 4.2 *In the specific case of shared libraries, the autoscripts mechanism is triggered by the invocation of the debhelper tool* `dh_makeshlibs` *(mnemonic for "make shared library"), which takes care of adding the following two script snippets respectively to the* `postinst` *and* `postrm` *maintainer scripts:*

Listing 4.1: postinst-makeshlibs

```
if [ "$1" = "configure" ]; then
    ldconfig
fi
```

Listing 4.2: postrm-makeshlibs

```
if [ "$1" = "remove" ]; then
    ldconfig
fi
```

(The "if..then" guards are uninteresting internal details, which only ensure that the cache update mechanism is not triggered in corner case execution paths of the Debian installer `dpkg`*.)*

From the point of view of the package maintainer, the autoscript machinery can be completely ignored as long as all needs of writing maintainer scripts are addressed by `debhelper`. In all such cases the resulting maintainer scripts are entirely generated with no script code manually written by the maintainer, by just composing together sequentially one or more of the autoscript templates. To our ends, this means that we can restrict our analysis to the templates themselves, because they are either verbatim copied in the resulting scripts or—in the worst case scenario—filled using simple textual "holes" such as the current package names.

We investigated the `debhelper` code to extract the autoscripts template, finding 52 of them. All those templates are reported in Appendix A.1. Each one of these templates contains statements that are often jointly executed. For this reason they become special statements in the metamodel, as can be seen in Chapter 5. In this way maintainers, which are used to write scripts by means of Debhelper and the previously identified templates, will find suitable and familiar statements in the metamodel.

When the maintainer needs to add specific code to maintainer scripts, which is not provided by autoscript templates, `debhelper` enables mixing generated lines with lines written by hand. Skipping the details on how the maintainer achieves that, the resulting scripts are composed by (possibly alternated) sequences of generated and hand-written lines. All generated lines are tagged with specially-crafted comments, so that they are recognizable mechanically.

Example 4.3 *The* `postrm` *script of the* `libxenomai1` *library package in the considered Debian Lenny snapshot reads:*

```
1  #!/bin/sh
2  set -e
3
4  case "$1" in
5    purge | remove)
6    [ ! -L /etc/udev/rules.d/xenomai.rules ] || rm /etc/udev/rules.d/xenomai.rule\
7  s
8    ;;
9  esac
10
11 # Automatically added by dh_makeshlibs
12 if [ "$1" = "remove" ]; then
13         ldconfig
14 fi
15 # End automatically added section
```

In the script, we can recognize that the final part (lines 11-15) is generated using autoscripts to update the already discussed linked cache, preceded by a hand written part (lines 4–9) contributed by the package maintainer. The remaining lines are either inert blank lines or the common maintainer script "header" (lines 1–2), prescribed by the Debian policy.

Starting from the non-empty maintainer scripts extracted from Debian Lenny (summing up to 25'440 scripts), we analyzed how many of them are *entirely* composed by lines generated using the autoscript mechanism. Also, we produced a "filtered" version of all the remaining maintainer scripts (i.e., those that contain at least *some* line written by hand by the package maintainer) which has been analyzed later on in more details.

The summary of generated (part of) maintainer scripts is as follows (LOC in table is for lines of code):

	n. of scripts	*LOCs*
non-blank	25'440 (100%)	386'688 (100%)
generated (non-blank)	16'348 (64.3%)	162'074 (41.9%)
by hand (non-blank)	9'061 (35.6%)	224'614 (58.1%)

About 2/3 of all the maintainer scripts are composed only of lines generated using the autoscript mechanism.

4.1.2 Analysis of scripts "by hand"

The scripts that survived to the previous phases are 9061. These scripts are analyzed "by hand". The idea of this analysis is to find additional templates or additional statements that should be considered when defining the metamodel.

The analysis "by hand" has been performed as follows:

1. all the scripts that survived to the previous pruning phases are clustered in groups, where a group collects scripts that contain exactly the same statements. For each group we then selected one representative. Table 4.1 shows an excerpt of the groups that we identified, ordered by occurrence. Therefore, the second column shows the occurrence while the third column contains the name of the script representative of the group.

Group	Occurrence	Representative script name
G1	93	libk/libkpathsea4_2007.dfsg.2-4_amd64.deb.preinst
G2	54	d/dict-freedict-swe-eng_1.3-4_all.deb.postinst
G3	54	d/dict-freedict-fra-deu_1.3-4_all.deb.postrm
G4	35	j/jabber-jud_0.5-3+b1_amd64.deb.preinst
G5	35	g/gauche-c-wrapper_0.5.4-2_amd64.deb.postinst
G6	33	w/wogerman_2-25_all.deb.config
G7	31	m/mii-diag_2.11-2_amd64.deb.prerm
G8	30	libs/libsocket6-perl_0.20-1_amd64.deb.postrm
G9	28	m/myspell-it_2.4.0-3_all.deb.postinst
G10	27	libp/libpaper1_1.1.23+nmu1_amd64.deb.postinst
G11	26	i/ibritish_3.1.20.0-4.4_amd64.deb.config
G12	24	e/education-geography_0.837_amd64.deb.postrm
G13	24	e/education-development_0.837_amd64.deb.postinst
G14	23	libt/libtcd-dev_2.2.2-1_amd64.deb.postinst
G15	22	g/gpppon_0.2-4+b1_amd64.deb.postinst
G16	22	m/myspell-en-us_2.4.0-3_all.deb.postrm
G17	21	w/webmagick_2.02-8.3_all.deb.preinst
G18	20	libg/libghc6-stream-doc_0.2.2-2_all.deb.postrm
G19	20	g/guidedog_1.0.0-4_amd64.deb.prerm
G20	20	libg/libghc6-hgl-doc_3.2.0.0-3_all.deb.postinst
G21	18	j/junior-system_1.13_all.deb.postinst
G22	18	j/junior-kde_1.13_all.deb.postrm
G23	18	libs/libsnmp15_5.4.1 dfsg-11_amd64.deb.prerm
G24	17	j/jabber-common_0.5_all.deb.prerm
G25	15	g/gauche_0.8.13-1_amd64.deb.postrm
G26	15	g/gnumeric_1.8.3-5_amd64.deb.config
G27	14	s/science-engineering_0.3_all.deb.postrm
G28	14	g/gtalk_0.99.10-12_amd64.deb.prerm
G29	14	m/mii-diag_2.11-2_amd64.deb.postinst
G30	14	s/science-engineering_0.3_all.deb.postinst
G31	11	d/debian-reference-es_2.17_all.deb.postinst
...	...	...

Table 4.1: Excerpt of the obtained groups

For instance, the group G14, reported in Listing 4.3, contains 23 scripts (i.e., 22 scripts are absolutely equal to the script `libtlibtcd-dev_2.2.2-1_amd64.deb.postinst` that is the representative of this group).

Listing 4.3: Script `libtlibtcd-dev_2.2.2-1_amd64.deb.postinst`

```
set -e
case "$1" in
    configure)
    ;;
    abort-upgrade|abort-remove|abort-deconfigure)
    ;;
    *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac
exit 0
```

The idea of this step is that, referring to Table 4.1 group G1, 92 scripts are absolutely equal to the script `libk/libkpathsea4_2007.dfsg.2-4_amd64.deb.preinst` and then this script probably contains a potential template.

2. As can been seen in Listing 4.3 the script contains specific statements that can be removed in order to find a template, such as `exit 0` or `set -e`. Therefore, in this step we identify templates from these recurrent scripts. Listing 4.4 shows *Template2* obtained from the script `libtlibtcd-dev_2.2.2-1_amd64.deb.postinst`, group G14.

Listing 4.4: Template2

```
case "$1" in
    configure)
    ;;
    abort-upgrade|abort-remove|abort-deconfigure)
    ;;
    *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac
```

Sometimes we identified different templates inside the same script.

3. The next step consists in identifying the occurrence of the template in the collection of 9061 scripts. For instance the occurrence of *Template2* is 69. Table 4.2 shows an excerpt of the templates occurrences. More precisely the first column describes the occurrence of the template, the second one shows the template name, the third column contains the name of the script that originated the template and the last column shows the group in which the script belongs.

4. Once Templates identified (we identified 116 templates) and the occurrences for each single template identified, the next step consists in identifying similarities among templates in order to collect them in classes. In fact, we recall that the occurrences are calculated with exact matching and that a white space can also compromise the matching. For instance Template2 and Template4, shown in Listing 4.4 and Listing 4.5 respectively, that differ only in the exit statement belong to the same class. The result of this step is the identification of 10 classes.

Listing 4.5: Template4

```
case "$1" in
    configure)
```

Occurrence	Template	Origin Group
93	Template1	G1
97	Template1a	G1
97	Template1b	G1
97	Template1c	G1
97	Template1d	G1
97	Template1e	G1
69	Template2	G14
16	Template3	G39
41	Template4	G15
31	Template5	G8
...	...	...

Table 4.2: Excerpt of the occurrences of the Templates

```
    ;;
    abort-upgrade|abort-remove|abort-deconfigure)
    ;;
    *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 0
    ;;
esac
```

These 10 classes collect 1340 scripts.

5. The next step consists in analyzing each class in order to understand how to deal with this kind of scripts. In other words, we have to understand whether the already identified statements are sufficient or whether new statements are required.

   Class 1

   Class 1 collects `postinst` scripts that are used to perform actions required after the installation of a package. Examples of scripts belonging to this class are Template2 and Template4. The possible actions performed are: `configure`, `abort-upgrade`, `abort-remove` and `abort-deconfigure`. This class collects scripts that enclose these actions in a `case` statement. Therefore, the metamodel should contain a special `case` statement specialized with these possible actions. Actually, these scripts do absolutely nothing, as can be seen in Listing 4.5, but it is important to note that this kind of case statement is very recurrent also in more complex scripts. The same observation holds for Classes 2, 3, and 4.

   Class 2

   Similarly to Class 1, Class 2 consists in a case statement that contains the different actions that can be performed. Class 2 collects `postrm` scripts and then the possible actions are: `purge`, `remove`, `upgrade`, `failed-upgrade`, `abort-install`, `abort-upgrade` and `disappear`. Morover, in this case, the metamodel will have a specialized `case` statement with the possible actions.

   Listing 4.6 shows Template3 as an example of a template belonging to this class.

Listing 4.6: Template3

```
case "$1" in
    purge|remove|upgrade|failed-upgrade|abort-install|abort-upgrade|disappear)
    ;;
    *)
        echo "postrm called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac
```

Template	Template occurrence
Template2	69
Template4	41
Template9	4
Template23	3
Template24	70
Template25	54
Template29	20
Template31	41
Template36	7
Template38	5
Template40	6
Template41	3
Template43	3
Template44	3
Template46	3
Template47	2
Template48	1
Total	**335**

Table 4.3: Class 1

Template	Template occurrence
Template3	16
Template5	31
Template6	7
Template51	54
Template54	15
Template55	11
Template56	10
Template58	17
Template59	5
Template60	5
Template62	5
Template63	4
Template67	4
Template68	4
Template69	3
Template73	3
Template76	3
Total	**197**

Table 4.4: Class 2

Template	Template occurrence
Template81	48
Template82	48
Template83	46
Template84	20
Template86	20
Template88a	49
Template95	3
Template96	3
Template97	3
Total	**240**

Table 4.5: Class 3

Template	Template occurrence
Template78	37
Template79	22
Template80	23
Template89	5
Template90	4
Template93	3
Template94	3
Template98	3
Total	**97**

Table 4.6: Class 4

Class 3

Analogously to Class 1 and Class 2, Class 3 is the class for the `prerm` scripts. The actions are: `remove`, `upgrade`, `deconfigure` and `failed-upgrade`. The metamodel will have a specialized `case` statement with the possible actions.

Listing 4.7 shows Template81 as an example of a template belonging to this class.

Listing 4.7: Template81

```
case "$1" in
    remove|upgrade|deconfigure)
        ;;
    failed-upgrade)
        ;;
    *)
        echo "prerm called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac
```

Class 4

Analogously to Class 1, Class 2, and Class 3, Class 4 is the class for the `preinst` scripts. The actions are: `install`, `upgrade` and `abort-upgrade`. The metamodel will have a specialized `case` statement with the possible actions.

Listing 4.8 shows Template78 as an example of a template belonging to this class.

Listing 4.8: Template78

```
case "$1" in
    install|upgrade)
    ;;
    abort-upgrade)
    ;;
    *)
```

Template	Template occurrence
Template7	21
Template8	18
Template10	4
Template11	5
Template14	4
Template15	3
Template18	5
Template20	33
Template21	3
Template26	34
Template42	3
Total	**133**

Table 4.7: Class 5

Template	Template occurrence
Template27	27
Template17	3
Total	**30**

Table 4.8: Class 6

```
7         echo "preinst␣called␣with␣unknown␣argument␣\‘$1’" >&2
8         exit 1
9     ;;
10 esac
```

Class 5

Class 5 collects `postinst` scripts. As can be seen in Listing 4.9, this class of scripts is already covered by the template of `debhelper` shown in Listing A.22.

Listing 4.9: Template7

```
1 if [ "$1" = "configure" ]; then
2     ldconfig
3 fi
```

Class 6

Actually, this class of scripts collects scripts with only one statement, as can be seen in Listing 4.10 that shows Template27. The only one lesson that we can learn is that the statement is a statement that potentially modify the environment.

Listing 4.10: Template27

```
1 . /usr/share/debconf/confmodule
```

Class 7

This class contains `postinst` and `postrm` scripts. What we can see is that the case structure highlighted by classes 1,2,3 and 4 is also respected in these scripts. Furthermore, this class contains a very general case statement. This means that for the case statement we also need a general case statement that can be used for very different cases.

Listing 4.11: Template28

```
1 if [ -d /etc/cdd -a -f /etc/cdd/cdd.conf ] ; then
2     if [ -d /etc/cdd/education -a -f /etc/cdd/education/education.conf ] ; then
3         test -x /usr/sbin/cdd-update-menus && /usr/sbin/cdd-update-menus -d
              ↪education
4     if [ _"$DEBCONF_REDIR" = _"" ]; then
```

Template	Template occurrence
Template28	24
Template30	18
Template32	14
Template34	11
Template66	6
Template70	3
Template71	8
Template74	34
Template75	9
Total	**127**

Table 4.9: Class 7

Template	Template occurrence
Template12	15
Template37	18
Template39	5
Template61	16
Template64	4
Template65	21
Template77	22
Total	**101**

Table 4.10: Class 8

```
        . /usr/share/debconf/confmodule
            db_version 2.0
    fi
    . /etc/cdd/cdd.conf
    if [ -s /etc/cdd/education/education.conf ] ; then . /etc/cdd/education/
education.conf ; fi
    case "$1" in
        abort-deconfigure|abort-remove|abort-upgrade)
        ;;
            configure|upgrade)
        db_get "shared/education-config/usermenus" || true
        case "$RET" in
                "Each package installation")
                /usr/sbin/cdd-update-usermenus education
                    ;;
                "End of installation")
                    touch /var/run/education-config.usermenu
                    ;;
            esac
        ;;
        *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 1
        ;;
    esac
    db_stop
    fi
fi
```

Class 8

This class of `postinst` and `postrm` scripts is composed of an `if` control statement that regulates the update of the menu. This class is very similar to the templates of `debhelper` shown in Listings A.26 and A.27.

Listing 4.12: Template12

```
if test -x /usr/bin/update-menus; then update-menus; fi
```

Template	Template occurrence
Template87	5
Template91	4
Total	**9**

Table 4.11: Class 9

Template	Template occurrence
Template100	68
Total	**68**

Table 4.12: Class 10

Class 9

Nothing should be added to the metamodel in order to deal with this class of scripts. An example of scripts belonging to this class is in Listing 4.13.

Listing 4.13: Template87

```
if [ "$1" = "configure" ]; then
    fontdirs="koi8-r.misc koi8-r.75dpi koi8-r.100dpi"
    for currentdir in $fontdirs; do
    longdir=/usr/lib/X11/fonts/$currentdir
    if [ -d $longdir ]; then
        if [ $(find $longdir| egrep -v '/(fonts.dir|fonts.alias|encodings.dir)$'|
            ↪wc -l) -eq 1 ]; then
        rm $longdir/fonts.dir 2>/dev/null || true
        rm $longdir/fonts.alias 2>/dev/null || true
        rm $longdir/encodings.dir 2>/dev/null || true
        rmdir $longdir
        fi
    fi
    done
fi
```

Class 10

This class is not really a class since it is composed of only one template, i.e., Template 100 shown in Listing 4.14.

Listing 4.14: Template100

```
if ( -e $script ){
    require $script;
    dc_debconf_select($class);
}
```

6. Other 307 scripts have been analyzed in the previous step even thought they cannot be classified in a Class.

7. The last step has been the analysis of scripts "by hand" with occurrence 1 in order to understand whether they are already covered or whether they contain statements that we are not able to deal with. In this last case we analyzed other 43 scripts.

4.2 Maintainer script analysis: RPM-based distributions

RPM (Red Hat Package Manager) is one of the most common software package manager used for Linux distributions. Although RPM was originally designed to work with Red Hat Linux,

it also works on other rpm-based distributions, such as Mandriva Linux, Fedora, Suse and Conectiva.

The `spec` file has one of the main important roles of the RPM's packaging building process. In fact, the spec file contains all the information needed to (i) compile the program and build source and binary rpms, and (ii) install and uninstall the program on the final user's machine. A `spec` file, as can be seen in the one reported in Listing 4.15, is divided into eight sections:

Preamble : it contains information that will be displayed when users request information about the package, such as the version number of the software, sources, patches, etc;

Prep : it is where the necessary preparations are made prior to the actual building of the software, such as the unpacking of the sources;

Build : this section contains commands required to compile the sources;

Install : this section is used to perform the commands required to actually install the software;

Install and Uninstall Scripts : this section consists of scripts that will be executed, on the user's system, when the package is actually installed or removed (like the maintainer scripts in Debian);

Verify Script : this is executed for verifying the proper installation of the package;

Clean Script : this is executed to clean things after the build;

File List : it consists of a list of files that will comprise the package;

Each section is denoted by a corresponding keyword like `%build` in Listing 4.15 which marks the beginning of the *build* section which contains the script executed by the RPM package manager to pack the `855resolution` software. It should be noted that, in the strictest sense of the word, these parts of the `spec` file are not scripts. For example, they do not start with the traditional invocation of a shell. However, the contents of each script section are copied into a file and executed by RPM as a full-fledged script.

Since we are interested in `install` and `uninstall` scripts, the rest of the section focuses on the *Install and Uninstall Scripts* section only. Similarly to Debian, in the RPM format there are four kind of scripts, each one meant to be executed at different stages of the package upgrade process. For further details, please refer to [Bai97]:

- `%pre` is executed before installation. It is not very common having RPM packages that require anything to be done prior to installation; none of the 350 packages that comprise Red Hat Linux Linux 4.0 make use of it;
- `%post` is executed after installation. A typical `%post` script consists of the `ldconfig` command which updates the list of available shared libraries after a new one has been installed. If a package uses a `%post` script to perform some function, quite often it will include a `%postun` script that performs the inverse of the `%post` script, after the package has been removed;
- `%preun` is executed before removing packages. This kind of script is used to prepare the system immediately prior the package deletion;

- `%postun` is executed after package deletions. Quite often, `%postun` scripts are used to run `ldconfig` to remove newly erased shared libraries from ld.so.cache. As highlighted before, these scripts typically do the inverse of `%post` ones.

Listing 4.15 shows an example of spec file, where we can see among the other information the %post and %preun scripts.

Listing 4.15: Example of a .spec file: 855resolution package

```
# $Id$
# Authority: matthias

Summary: Change video bios resolutions on laptops with Intel graphic chipsets
Name: 855resolution
Version: 0.4
Release: 4
License: Public Domain
Group: Applications/System
URL: http://perso.wanadoo.fr/apoirier/
Source0: http://perso.wanadoo.fr/apoirier/855resolution-%{version}.tgz
Source1: 855resolution.init
Source2: 855resolution.pm-hook
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root
# This utility doesn't make sense on other archs, those chipsets are i386 only
ExclusiveArch: i386

%description
This software changes the resolution of an available vbios mode. It is useful
when the native screen resolution isn't advertised as available by the video
bios by default.

It patches only the RAM version of the video bios so the new resolution is
lost after each reboot. If you want to have the resolution set after each
boot, then you need to edit %{_sysconfdir}/sysconfig/855resolution.


%prep
%setup -n %{name}
# Add OPTFLAGS to CFLAGS
%{__perl} -pi -e 's|-Wall|-Wall \${OPTFLAGS}|g' Makefile


%build
%{__make} %{?_smp_mflags} OPTFLAGS="%{optflags}"


%install
%{__rm} -rf %{buildroot}

# Manually install the binary
%{__install} -D -m 0755  855resolution %{buildroot}%{_sbindir}/855resolution

# Init script
%{__install} -D -m 0755 %{SOURCE1} \
    %{buildroot}%{_sysconfdir}/rc.d/init.d/855resolution

# Power Management hook, as 15 since video is 20 (for suspend to disk)
%{__install} -D -m 0755 %{SOURCE2} \
    %{buildroot}%{_sysconfdir}/pm/hooks/15resolution

# Default sysconfig entry.
%{__mkdir_p} %{buildroot}%{_sysconfdir}/sysconfig/
%{__cat} > %{buildroot}%{_sysconfdir}/sysconfig/855resolution << EOF
# Mode to overwrite (use "855resolution -l" to see all available modes)
MODE="49"
# Resolution to set (i.e. "1280 768", no "x", only a space as the separator)
RESOLUTION="1280 768"
EOF
```

```
60
61 %clean
62 %{__rm} -rf %{buildroot}
63
64
65 %post
66 if [ $1 -eq 1 ]; then
67     /sbin/chkconfig --add 855resolution
68 fi
69
70 %preun
71 if [ $1 -eq 0 ]; then
72     /sbin/chkconfig --del 855resolution
73 fi
74
75
76 %files
77 %defattr(-, root, root, 0755)
78 %doc CHANGES.txt LICENSE.txt README.txt
79 %config %{_sysconfdir}/pm/hooks/15resolution
80 %config %{_sysconfdir}/rc.d/init.d/855resolution
81 %config(noreplace) %{_sysconfdir}/sysconfig/855resolution
82 %{_sbindir}/855resolution
83
84
85 %changelog
86 * Thu Mar 23 2006 Matthias Saou <http://freshrpms.net/> 0.4-4
87 - Add pm hook script in order to fix suspend to disk resume, as the video BIOS
88   resolution needs to be overwritten before video is started upon resume too.
89   Thanks to Luke Hutchison for the script and the testing.
90
91 * Fri Mar 17 2006 Matthias Saou <http://freshrpms.net/> 0.4-3
92 - Release bump to drop the disttag number in FC5 build.
93
94 * Tue Jul  5 2005 Matthias Saou <http://freshrpms.net/> 0.4-2
95 - Make package ExclusiveArch i386, it doesn't make sense on other archs.
96 - Fix init script (add subsys lock) to not have it run on each runlevel change.
97 - Enable service by default : People who install this package want it!
98
99 * Mon Jul  4 2005 Matthias Saou <http://freshrpms.net/> 0.4-1
100 - Initial RPM release.
```

Similarly to Debian, Fedora, an RPM-based Linux distribution[5], also makes use of templates for the maintainer scripts[6]. Such templates, called autoscripts, are reported in Appendix A.2. Mandriva[7], another RPM-based Linux distribution, makes use of the macros[8] reported in Appendix A.3.

In this section we analyze the Fedora distribution by considering the `.spec` which can be downloaded at `http://svn.rpmforge.net/svn/trunk/rpms/`. The available `.spec` files are 4'704, and considering that each of them can contain four kinds of scripts, the potential universe that has to be analyzed consists of 4'704*4=18'816 scripts. Actually, the scripts that are present in this set are 2'038, that is approximatively 10.8% of 18'816. These scripts are divided as follows: 81 %`pre`, 911 %`post`, 234 %`preun`, and 812 %`postun`. We extracted the four kinds of scripts from each spec file and, in order to make the analysis, we created four new files containing the scripts; the name of the generated files follows the convention <`file.spec`>.<`script_type`>.

Unfortunately, in this case we are not able to identify scripts that are generated from helpers, since we have not found particular comments that help in identifying the generated code. For this reason, we performed analysis "by hand", similarly to what is described for Debian in

[5] Fedora Project Web site: `http://fedoraproject.org`
[6] Fedora ScriptletSnippets: `http://fedoraproject.org/wiki/Packaging/ScriptletSnippets`
[7] Mandriva Web site: `http://mandriva.org/`
[8] Mandriva RPM HOWTO: `http://wiki.mandriva.com/en/Development/Howto/RPM`

Group	Occurrence	Representative script name
G1	295	./evolution-rss/evolution-rss.spec.postun
G2	277	./libewf/libewf.spec.post
G3	147	./gtkhtml3/gtkhtml3.spec.postun
G4	143	./gtkhtml3/gtkhtml3.spec.post
G5	30	./gtranslator/gtranslator.spec.postun
G6	22	./clamav/clamav.spec.post
G7	21	./gpgme/gpgme.spec.postun
G8	20	./kernel-module-madwifi/kernel-module-madwifi.spec.postun
G9	17	./kernel-module-madwifi/kernel-module-madwifi.spec.post
G10	14	./dkms-lirc/dkms-lirc.spec.preun
G11	14	./dkms-lirc/dkms-lirc.spec.post
G12	13	./epiphany/epiphany.spec.post
G13	12	./camorama/camorama.spec.postun
G14	10	./straw/straw.spec.post
G15	10	./avidemux2/avidemux2.spec.postun
G16	9	./muine/muine-0.4.spec.post
G17	9	./camorama/camorama.spec.post
G18	9	./avidemux2/avidemux2.spec.post
G19	9	./muine/muine-0.4.spec.postun
G20	9	./hello/hello.spec.preun
G21	7	./gtranslator/gtranslator.spec.post
G22	6	./wol/wol.spec.post
G23	5	./liferea/liferea.spec.postun
G24	5	./gstreamer-ffmpeg/gstreamer-ffmpeg-0.8.spec.postun
G25	5	./kernel-module-hostap/kernel-module-hostap.spec.postun
G26	5	./liferea/liferea.spec.post
G27	5	./gstreamer-ffmpeg/gstreamer-ffmpeg-0.8.spec.post
G28	5	./kernel-module-hostap/kernel-module-hostap.spec.post
G29	4	./rpm5/rpm5.spec.postun
...	...	...

Table 4.13: Excerpt of the obtained groups for Fedora

Section 4.1.2. Then, all the scripts are clustered in groups, where a group collects scripts that contain exactly the same statements. For each group we then selected one representative. Table 4.13 shows an excerpt of the groups that we identified, ordered by occurrence.

The group G1 occurs 295 times and it is particularly interesting to see that this group collects scripts that are, as can be seen in Listing 4.16, exactly the template of Fedora for shared libraries reported in Listing A.73.

Listing 4.16: Group G1

```
1 %postun -p /sbin/ldconfig
```

The group G2 occurs 277 times and in this case this group collects scripts that are generated, as can be seen in Listing 4.17, from the Fedora template for shared libraries reported in Listing A.72.

Listing 4.17: Group G2

```
1 %post -p /sbin/ldconfig
```

The occurrence of group G3 (see Listing 4.18) is 147 and is a simple modification of Fedora template for shared libraries in Listing A.71.

Listing 4.18: Group G3

```
1 %postun
2 /sbin/ldconfig 2>/dev/null
```

Analogously, group G4 (see Listing 4.19) is a simple modification of the template for shared libraries in Listing A.70.

Listing 4.19: Group G4

```
%post
/sbin/ldconfig 2>/dev/null
```

Group G5 (see Listing 4.20) is exactly equal to template in Listing A.60 of Fedora for Gnome and KDE environments that use the scrollkeeper cataloging system to keep track of documentation installed on the system.

Listing 4.20: Group G5

```
%postun
scrollkeeper-update -q || :
```

Group G6 (see Listing 4.21) is exactly equal to template in Listing A.70.

Listing 4.21: Group G6

```
%post
/sbin/ldconfig
```

Group G7 (see Listing 4.22) is exactly equal to template in Listing A.71.

Listing 4.22: Group G7

```
%postun
/sbin/ldconfig
```

Group G12 (see Listing 4.23) is the concatenation of two templates: Listing A.91 and Listing A.60, respectively.

Listing 4.23: Group G12

```
%post
export GCONF_CONFIG_SOURCE="$(gconftool-2␣--get-default-source)"
gconftool-2 --makefile-install-rule %{_sysconfdir}/gconf/schemas/%{name}.schemas &>/dev/
    ↪null
scrollkeeper-update -q || :
```

Group G13 (see Listing 4.24) is a simple modification of Fedora template in Listing A.60.

Listing 4.24: Group G13

```
%postun
scrollkeeper-update -q
```

Group G14 (see Listing 4.25) is the template in Listing A.91.

Listing 4.25: Group G14

```
%post
export GCONF_CONFIG_SOURCE="$(gconftool-2␣--get-default-source)"
gconftool-2 --makefile-install-rule %{_sysconfdir}/gconf/schemas/%{name}.schemas &>/dev/
    ↪null
```

Group G15 (see Listing 4.26) is a customization of the template in Listing A.62.

Listing 4.26: Group G15

```
%postun
update-desktop-database %{_datadir}/applications &>/dev/null || :
```

Group G16 (see Listing 4.27) is a simple modification of the template for shared libraries in Listing A.70.

Group	Occurrence	Template
G1	295	Listing A.73
G2	277	Listing A.72
G3	147	Listing A.71
G4	143	Listing A.70
G5	30	Listing A.60
G6	22	Listing A.70
G7	21	Listing A.71
G12	13	Listing A.91 and Listing A.60
G13	12	Listing A.60
G14	10	Listing A.91
G15	10	Listing A.62
G16	9	Listing A.70
G17	9	Listing A.91 and Listing A.60
G18	9	Listing A.61
G19	9	Listing A.71
G20	9	Listing A.58
G21	7	Listing A.59
G22	6	Listing A.57

Table 4.14: Scripts that can be generated from templates

Listing 4.27: Group G16

```
%post
/sbin/ldconfig &>/dev/null
```

Analogously to Group 12, Group G17 (see Listing 4.28) is the concatenation of two templates: Listing A.91 and Listing A.60, respectively.

Listing 4.28: Group G17

```
%post
export GCONF_CONFIG_SOURCE="$(gconftool-2␣--get-default-source)"
gconftool-2 --makefile-install-rule %{_sysconfdir}/gconf/schemas/%{name}.schemas &>/dev/
    ↪null
scrollkeeper-update -q
```

Group G18 is equal to template in Listing A.61.

Listing 4.29: Group G18

```
%post
update-desktop-database %{_datadir}/applications &>/dev/null || :
```

Group G19 (see Listing 4.30) is a simple modification of the template for shared libraries in Listing A.71.

Listing 4.30: Group G19

```
%postun
/sbin/ldconfig &>/dev/null
```

Group G20 (see Listing 4.31) is equal to template in Listing A.58.

Listing 4.31: Group G20

```
%preun
/sbin/install-info --delete %{_infodir}/%{name}.info.gz %{_infodir}/dir
```

From this first analysis we can say that more than 50% of the scripts can be generated by templates. Table 4.14 summarizes these scripts and highlights involved templates.

Furthermore, probably new templates can be generated from the groups G8-G11 that apparently do not match with any template, even though they have good occurrences. They are reported in the following Listing:

Template	Occurrence	Group
T1	341	G1
T2	277	G2
T3	164	G3
T4	162	G4
T5	42	G5
T6	73	G6
T7	56	G7
T8	27	G8
T9	25	G9
T10	21	G10
T11	17	G11
T12	15	G12
T13	25	G13
T14	28	G14
T15	38	G15
T16	10	G16
T17	51	G20
T18	13	G21
T19	7	G22

Table 4.15: Matches of templates identified by the groups in Table 4.13

Listing 4.32: Group G8

```
%postun
/sbin/depmod -ae %{kversion}-%{krelease} || :
```

Listing 4.33: Group G9

```
%post
/sbin/depmod -ae %{kversion}-%{krelease} || :
```

Listing 4.34: Group G10

```
%preun
# Remove all versions from DKMS registry
dkms remove -m %{dkms_name} -v %{dkms_vers} %{?quiet} --all || :
```

Listing 4.35: Group G11

```
%post
# Add to DKMS registry
dkms add -m %{dkms_name} -v %{dkms_vers} %{?quiet} || :
# Rebuild and make available for the currenty running kernel
dkms build -m %{dkms_name} -v %{dkms_vers} %{?quiet} || :
dkms install -m %{dkms_name} -v %{dkms_vers} %{?quiet} --force || :
```

Once the scripts previously identified deleted, we continued the analysis, the next step being to use the identified templates in order to check their matches as part of the code of a script. The obtained matches are reported in Table 4.15.

It is important to note that groups G17, G18, and G19 do not define templates since they are already covered by previously reported templates.

Since the matches are always exact, we performed another step of analysis. In what follows we report the analysis we performed by manually inspecting the scripts and manually identifying the match. We defined other templates that complement the already defined templates. Table 4.16 shows the matches of these new templates together with the reference with the already defined templates if this relation exists. If the new templates cannot be considered a refinement of the previously presented templates but are captured by Fedora templates reported in Appendix A.2, a link to the corresponding listing is provided. Finally, the Reference field contains the string **New** if the template is identified by the analysis.

Refinement	Occurrence	Reference
Listing 4.36	31	T12
Listing 4.38	16	**New**
Listing 4.39	12	G8 and G9
Listing 4.40	59	Listing A.69
Listing 4.41	11	**New**
Listing 4.42	2	**New**
Listing 4.43	5	Listing A.74, A.75, and A.75
Listing 4.44	344	Listings A.74, A.75, and A.75
Listing 4.45	12	Listings A.63 and A.63
Listing 4.46	42	Listings A.89 and A.90
Listing 4.47	12	Listings A.68 and A.68
Listing 4.48	7	**New**
Listing 4.49	3	**New**
Listing 4.50	4	**New**

Table 4.16: Refinement of the analysis

Listing 4.36 shows the template obtained by suitably modifying the template T12.

Listing 4.36: Template 20

```
export GCONF_CONFIG_SOURCE="$(gconftool-2␣--get-default-source)"
gconftool-2 --makefile-uninstall-rule %{_sysconfdir}/gconf/schemas/%{name}.schemas &>/
    ↪dev/null || :
```

The analysis found a modification of the scripts by the addition of the line of code shown in Listing 4.37.

Listing 4.37: Template 20

```
killall -HUP gconfd -2 ||:
```

Listing 4.38 shows a new identified template.

Listing 4.38: Template 21

```
%postun
%{register} &>/dev/null || :
```

Listing 4.39 is related to groups G8 and G9.

Listing 4.39: Template 22

```
depmod -ae -F /boot/System.map-%{kernel} %{kernel} >/dev/null
```

Listing 4.40 reports a different version of the Fedora template in Listing A.69. However, the %pre template for user adding in Listing A.69 is not perfect as testified by the absence of exact matching. In fact, by hand we identified other 54 semantically equivalent scripts. The template should probably be split in different templates, i.e., one for user adding, one for group adding, etc.

Listing 4.40: Example of modification of the template Listing A.69

```
%pre
/usr/sbin/groupadd -g 37 rpm  > /dev/null 2>&1
/usr/sbin/useradd  -r -d /var/lib/rpm -u 37 -g 37 rpm -s
/sbin/nologin  > /dev/null 2>&1 exit 0
%endif
```

In line with this, we propose the new templates in Listings 4.41 and 4.42 for %postun scripts managing the user deleting and group deleting, respectively. The found matches are 11 for Listing 4.41 and 2 for Listing 4.41.

Listing 4.41: Template 24

```
%postun
if [ $1 -eq 0 ]; then
    /usr/sbin/userdel USERNAME 2>/dev/null || :
fi
```

Listing 4.42: Template 25

```
%postun
if [ $1 -eq 0 ]; then
    /usr/sbin/groupdel USERNAME 2>/dev/null || :
fi
```

Listing 4.43 is related to the Fedora template in Listing A.74.

Listing 4.43: Template 26

```
%post
/sbin/chkconfig --add nagios
```

Listing 4.44 is related to the previous template and then to the Fedora templates as well in Listing A.74, A.75, and A.76.

Listing 4.44: Template 27

```
# This adds the proper /etc/rc*.d links for the script
/sbin/chkconfig --add <script>
```

Listing 4.45 is related to the Fedora templates in Listing A.63 and Listing A.64.

Listing 4.45: Template 28

```
update-mime-database %{_datadir}/mime &>/dev/null || :
```

Listing 4.46 is related to the Fedora templates in Listing A.89 and Listing A.90.

Listing 4.46: Template 29

```
touch --no-create %{_datadir}/icons/hicolor
if [ -x %{_bindir}/gtk-update-icon-cache ] ; then
  %{_bindir}/gtk-update-icon-cache --quiet %{_datadir}/icons/hicolor || :
fi
```

Listing 4.47 is related to the Fedora templates in Listing A.67 and Listing A.68.

Listing 4.47: Template 30

```
if [ -x %{_bindir}/fc-cache ] ; then
  %{_bindir}/fc-cache %{_datadir}/fonts || :
fi
```

Finally, Listings 4.48, 4.49, and 4.50 show new templates that we propose for managing the %post, %preun, and %postun alternatives, respectively.

Listing 4.48: Template 31

```
%post
if [ -x /usr/sbin/alternatives ]; then
    /usr/sbin/alternatives --install %{_bindir}/%{name1} %{name1}  %{_bindir}/%{name2}
fi
```

Listing 4.49: Template 32

```
%preun
if [ $1 -eq 0 ]; then
    if [ -x /usr/sbin/alternatives ]; then
        /usr/sbin/alternatives --remove %{name1} %{_bindir}/%{name2}
    fi
fi
```

Listing 4.50: Template 33

```
%postun
if [ $1 -eq 0 ]; then
    if [ -x /usr/sbin/alternatives ]; then
        /usr/sbin/alternatives --remove %{name1} %{_bindir}/%{name2}
    fi
fi
```

To summarize, this analysis demonstrates that, by means of templates, 1'962 scripts among the 2'038 that constitute our universe of scripts can be automatically generated for sure (∼93,6%). Please also remember that the "potential" amount of scripts, as described at the beginning of this section, is the number of spec files multiplied by 4 (that is the number of different kinds of scripts). Then the total amount of potential scripts is 18'816, and the remaining scripts, which are 76, represent the 0,4% of the total amount. Furthermore, a large part of the remaining 76 scripts is simply a combination of some customized templates or parts of templates that could be modeled by means of more simple statements. These remaining 76 scripts can be found at `http://www.mancoosi.org/deliverables` together with the initial scripts and scripts we defined for performing the analysis.

One very interesting note is the presence of scripts that contain conversions from an old format to a new one. An example of these scripts is in Listing 4.51 that reports the script `sendmail/sendmail.spec.post`. This kind of scripts raises problems related to the roll-back support and represents one of the main aspects covered by the Mancoosi deliverable D3.2.

Listing 4.51: `sendmail/sendmail.spec.post` script

```
%post
#
# Convert old format to new
#
if [ -f /etc/mail/deny ] ; then
    cat /etc/mail/deny | \
    awk 'BEGIN{ print "# Entries from obsoleted /etc/mail/deny"} \
          {print $1" REJECT"}' >> /etc/mail/access
    cp /etc/mail/deny /etc/mail/deny.rpmorig
fi
for oldfile in relay_allow ip_allow name_allow ; do
    if [ -f /etc/mail/$oldfile ] ; then
    cat /etc/mail/$oldfile | \
        awk "BEGIN { print \"# Entries from obsoleted /etc/mail/$oldfile
\" ; } \
          { print \$1\" RELAY\" }" >> /etc/mail/access
    cp /etc/mail/$oldfile /etc/mail/$oldfile.rpmorig
     fi
done
```

4.3 Stemming out the elements to be modeled

By considering the outcomes of the analysis presented up to now, in this section we stem out the fundamental concepts building up maintainer scripts and hence, have to be properly formalized in terms of metamodels. The first observation is that the script statements can be classified in:

- *file system* statement, that are statements that work on the file system. They can operate with both files and directory and the different actions they can perform are addition, deletion and modification. Examples of these statements are `open/close` of files, `rm/copy/mv/mkdir` of files and directories, `ln` for creating symbolic links, etc;

- *environment* statements, they modify services, shared libraries and modules. Examples are `ldconfig`, `debmod`, `update_menus`, etc;
- *package settings* statements, they affect the package settings and configurations. Examples are `apache-conf`, `sendmailconfig`, etc.

In addition, there are also *control* statements that control the order in which statements are run. The recurrent control statements that we discovered are:

- *if*: this is used to decide whether to do something at a special point, or to decide between two courses of action;
- *case*: this is another form conditional statement. It is well structured, but can only be used in certain cases where: (i) only one variable is tested and all branches must depend on the value of that variable; (ii) each possible value of the variable can control a single branch. A final default branch may optionally be used to trap all unspecified cases.

  As highlighted before when analyzing classes 1, 2, 3 and 4 we have four specialized case statements for managing these classes. In particular we have:

  - *case postinst*: the values are `configure`, `abort-upgrade`, `abort-remove`, `abort-deconfigure`;
  - *case postrm*: the values are `purge`, `remove`, `upgrade`, `failed-upgrade`, `abort-install`, `abort-upgrade`, `disappear`;
  - *case prerm*: the values are `remove`, `upgrade`, `deconfigure`, `failed-upgrade`;
  - *case preinst*: the values are `install`, `upgrade`, `abort-upgrade`;
- *iterators*: iterators allow us to repeat designed statements for each element of a collection. The collections that we identified are:
  - *directories*: this allows us to iterate on each element of a directory, such as files, subdirectories etc. Examples are:
    - `for currentdir in $fontdirs; do`$\cdots$ from the script x/xfonts-bolkhov-koi8r-misc_1.1.20001007-6_all.deb.postinst
    - `for dir in /var/spool/MailScanner` from the script m/mailscanner_4.68.8-1_all.deb.postinst
  - *lines of a file*: this allows us to iterate on each line of a file. Examples are:
    - `foreach $line (@data)` {$\cdots$ from the script sslapd_2.4.11-1_amd64.deb.postinst
    - `for file_to_remove in $log_file $pid_file ; do` $\cdots$ where `log_file=/var/log/wdm-errors` from the script w/wdm_1.28-3_amd64.deb.postrm
    - `while read LINE do` $\cdots$ from the script libc/libc6_2.7-16_amd64.deb.postinst.
  - *enumeration*: this iterator allows us to execute the body of the iterator for each element of the enumeration. Examples are:
    - `for db in ‘get_database_list‘; do`$\cdots$ from the script sslapd_2.4.11-1_amd64.deb.postinst
    - `for service in $check; do`$\cdots$ from the script libc/libc6_2.7-16_amd64.deb.postinst
  - *input parameters*: this iterator allows us to execute the body of the iterator for each input parameter. Examples are:

    * `while [ $# -gt 0 ]; do···` from the script t/tex-common_1.11.3_all.deb.postinst
    * `while [ -n "$1" ]; do···` from the script x/x11-common_7.3+18_all.deb.config
  - *word*: this iterator allows us to execute the body of the iterator a number of times equal to the length of the word. An example is:
    * `while ($len--) {···` from the script sslapd_2.4.11-1_amd64.deb.postinst.

Finally, we also add neutral statements, i.e., statements that neither modify the file system nor the configuration nor the package settings. They are:

- *messages*: they allow us to write messages to both standard output and error output. They are intended to cover statements like `echo`, `more` etc.
- *comments*: they allow us to write comments to the scripts.
- *exit*: this allows us to exit from the script. Two different exit values are allowed in order to represent success, 0, or failure 1.

4.4 Uncovered elements

An important output of the analysis is also the identification of what we are currently not able to model, and then to manage. In particular, we do not deal with the interaction with the user such that: `while (<STDIN>) {···` from the script `o/openssh-server_5.1p1-3_amd64.deb.postinst`.

However, it is important to note that the `RPM` install is designed to be run with no user interaction. This was a design decision taken prior the development of the `RPM` packaging system.

Chapter 5

MANCOOSI metamodels

The analysis described in the previous section has induced the definition of different metamodels which will be described throughout this chapter. In particular, 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 real systems. The metamodels which have been defined are the following:

- the *System Configuration metamodel*, which contains all the modeling constructs to specify the configuration of a given FOSS system in terms of installed packages, configuration files, services, filesystem state, etc.;
- the *Package metamodel*, which describes the relevant elements making up a software package. The metamodel also gives the possibility to specify the maintainer script behaviors which are currently ignored—beside mere execution—by existing package managers;
- 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 Fig. 5.1, *System Configuration* and *Package* metamodels have mutual dependencies, whereas the *Log* metamodel has only direct relations with both *System Configuration* and *Package* metamodels. In the rest of the chapter, such metamodels are described in more details and some explanatory models conforming to them are also provided. Moreover, in order to have a precise and formal definition of the metamodels, the KM3 [JB06] language and its tool support are used. KM3 is based on the same core concepts used in OMG/MOF [Obj03b] and EMF/Ecore [BSM+03] that are classes, attributes and references. The use of KM3 is mainly justified by its simplicity and flexibility to write metamodels and to produce domain-specific languages. A number of experimental KM3 metamodels have been specified from both academia and industry and are currently collected into a library that can be found at [ATL].

5.1 System Configuration metamodel

A system configuration is the composition of artifacts necessary to make computer systems perform their intended functions [DH07]. In this respect, the metamodel depicted in Fig. 5.2 specifies the main concepts which make up the configuration of a FOSS system. The equivalent

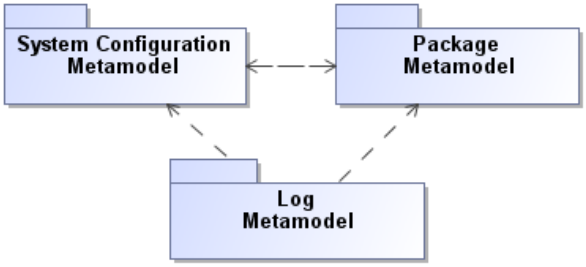

Figure 5.1: Metamodels and their inter-dependencies

KM3 textual specification is reported in Listing 5.1. The `Environment` metaclass enables the specification of loaded modules, shared libraries, and running process as in the sample configuration reported in Fig. 5.3. In such a model the reported environment is composed of the services `www`, and `sendmail` (see the instances `s1` and `s2`) corresponding respectively to the running web and mail servers.

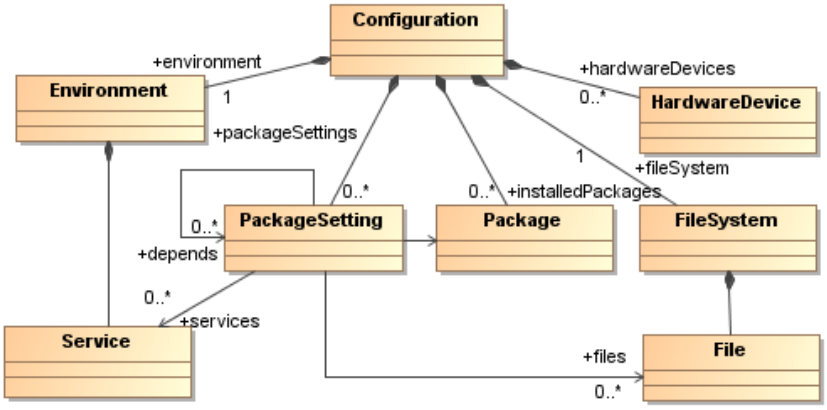

Figure 5.2: Graphical representation of the Configuration metamodel

All the services provided by a system can be used once the corresponding packages have been installed (see the association between the `Configuration` and `Package` metaclasses in Fig. 5.2) and properly configured (`PackageSetting`). Moreover, the configuration of an installed package might depend on other package configurations. For example, considering the PHP5 upgrade described in Section 2.2, the instances `ps1` and `ps2` of the `PackageSetting` metaclass in Fig. 5.3 represent the settings of the installed packages `apache2`, and `libapache-mod-php5`, respectively. The former depends on the latter (see the value of the attribute `depends` of `ps1` in Fig. 5.3) and both are also associated with the corresponding files which store their configurations. Note that at the level of package meta-information such a dependency should not be expressed, in spite of actually occurring on real systems. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by metamodeling.

Listing 5.1: KM3 specification of the Configuration Metamodel

```
1 class Configuration {
2     reference installedPackages[0-*] container : Package oppositeOf configuration;
3     reference runningServices[0-*] container : Service oppositeOf configuration;
4     reference packageSettings[0-*] container : PackageSetting oppositeOf configuration;
5     reference devices[0-*] container : HardwareDevice oppositeOf configuration;
6     reference fileSystem [1-1] container : FileSystem oppositeOf configuration;
7     reference environment [1-1]container : Environment oppositeOf configuration;
8 }
```

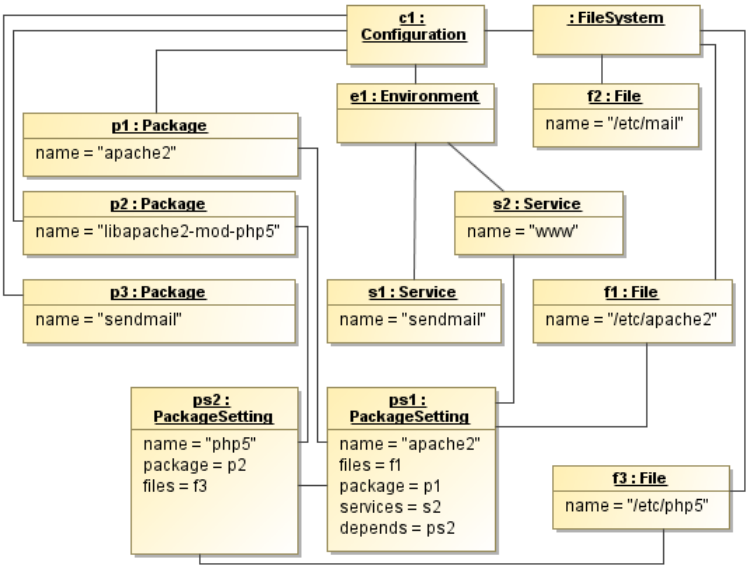

Figure 5.3: Sample Configuration model

```
9 ...
```

The configuration metamodel also gives the possibility to specify the hardware devices of a system by means of the `HardwareDevice` metaclass. This class has been considered to have a comprehensive metamodel able to cover both software and hardware concepts even though only the software aspects are taken into account in this document.

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

5.2 Package metamodel

The metamodel reported in Fig. 5.4 plays a key role in the overall upgrade simulation approach. In fact, in addition to the information already available in current package descriptions, the concepts captured by the metamodel enable the specification of the behavior of maintainer scripts. In this respect, the metaclass `Statement` in Fig. 5.4 represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration (`EnvironmentStatement`, `FileSystemStatement`, and `PackageSettingStatement`, respectively). For instance, the sample package model in Fig. 5.5 reports the scripts contained in the package `libapache-mod-php5` introduced in Section 2.2. For clarity of presentation, Fig. 5.5 contains only the relevant elements of the *postinst* and *prerm* scripts which are represented by the elements `pis1` and `prs1`, respectively.

According to the model in Fig. 5.5 the represented scripts update the configuration of the package `apache2` (see the element `ps1`) which depends on `libapache-mod-php5`. In particular, the element `upss2` corresponds to the statement `a2dismod` which disables the PHP5 module in the Apache configuration before removing the package `libapache-mod-php5` from the filesystem. This statement is necessary, otherwise inconsistent configurations can be reached like the one shown in Fig. 5.6. The figure reports the sample `Configuration2` which has been reached by removing `libapache-mod-php5` without changing the configuration of `apache2`.

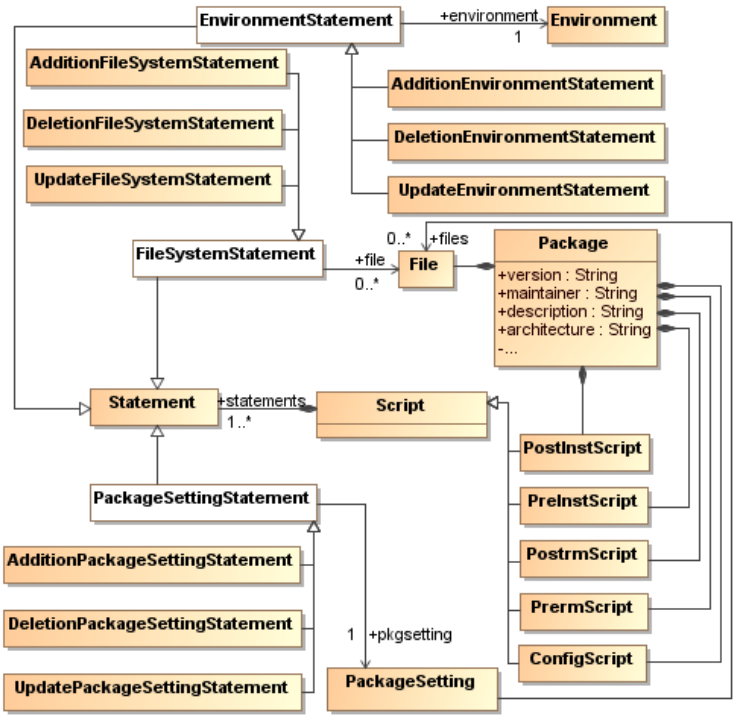

Figure 5.4: Overview of the Package metamodel

With respect to the underlying semantics of the elements contained in the package metamodel, such a configuration is not correct given that it contains a dependency between the `apache2` and `libapache-mod-php5` package settings, whereas only the package `apache2` is installed. Currently, the package managers are not able to predict inconsistencies like the one in Fig. 5.6 since they take into account only information about package dependencies and conflicts. The metamodel reported in Fig. 5.4 gives the possibility to specify an abstraction of the involved maintainer scripts which are executed during the package upgrades. This way, consistence checking possibilities are increased and trustworthy simulations of package upgrades can be operated.

In the rest of the section, the Package metamodel is described in details by discussing the KM3 specification of all the metaclasses previously outlined which underpin the script behavior specification.

5.2.1 Script metaclass

As discussed in Chapter 2, maintainer scripts can be executed at different stages of package upgrades. In this respect, they are classified into *preinst*, *postinst*, *prerm*, *postrm*, and *config*. The package metamodel fragment reported in Listing 5.2 takes into account this classification by extending the abstract metaclass `Script` with `PreinstScript`, `PostinstScript`, `PrermScript`, `PostrmScript`, and `ConfigScript`, respectively (see lines `10-28`). The statements building up a given script and its input parameters are captured in the metaclass `Script` by means of the references `statements`, and `inputParameters`, respectively (see lines `1-8`).

Listing 5.2: KM3 specification of the Script elements in the Package Metamodel

```
1 class Script extends NamedElement{
2     reference statements[0-*] container: Statement oppositeOf script;
```

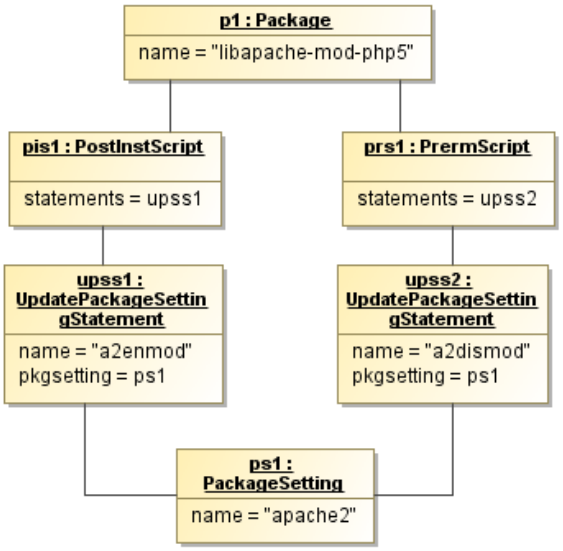

Figure 5.5: Sample Package model

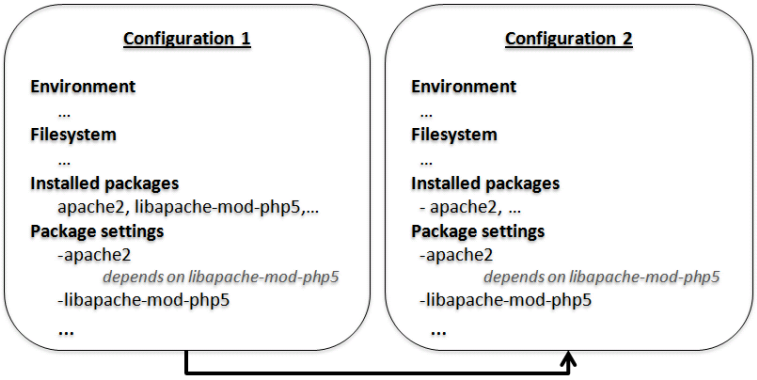

Figure 5.6: Incorrect package removal

```
3     reference inputParameters[0-*] container : InputParameter oppositeOf script;
4 }
5
6 class InputParameter extends NamedElement {
7     reference script : Script oppositeOf inputParameters;
8 }
9
10  class PreinstScript extends Script{
11         reference pkg : "Package" oppositeOf PreinstScript;
12 }
13
14 class PostinstScript extends Script{
15     reference pkg : "Package" oppositeOf PostinstScript;
16 }
17
18 class PrermScript extends Script{
19     reference pkg : "Package" oppositeOf PrermScript;
20 }
21
22 class PostrmScript extends Script{
23     reference pkg : "Package" oppositeOf PostrmScript;
24 }
25
```

```
26 class ConfigScript extends Script{
27     reference pkg : "Package" oppositeOf configScript;
28 }
```

5.2.2 Statement metaclass

According to the different configuration elements which can be affected by the execution of a given script statement, the abstract metaclass `Statement` in Listing 5.3 is specialized in different metaclasses that are `FileSystemStatement`, `EnvironmentStatement`, and `PackageSettingStatement` (see lines `7-17`). Moreover, each of them are in turn specialized for capturing additions, removals, and upgrades (see lines `24-42`). In particular, the statements which add, delete and modify the `FileSystem` (see Fig. 5.2) are respectively represented as `AdditionFileSystemStatement`, `DeletionFileSystemStatement` and `UpdateFileSystemStatement` instances. The shell commands `touch`, `rm` and `cp`, are sample instances of such metaclass.

The statements which modify the `Environment` of a given configuration are given in terms of instances of `EnvironmentStatement` specializations (see lines `31-35`). Shell commands like `install-menu`, `rmmod`, `ldconfig` of Linux distributions, can be respectively modeled as `AdditionEnvironmentStatement`, `DeletionEnvironmentStatement` and `UpdateEnvironmentStatement` instances.

As pointed out in the previous chapters, an installed package might depend on settings properly stored in dedicated configuration files (i.e., the service `apache2` depends on the configurations specified in the file `httpd.conf` usually stored in the `/etc/apache2` directory). The statements which modify such settings are modeled by means of instances of the `PackageSettingStatement` extensions (see lines `38-42`). Finally, maintainer scripts might contain statements which do not change the system configuration but are comments, emit messages, etc. Such cases can be specified by means of instances of the `NeutralStatement` metaclass (see line `21`).

Listing 5.3: KM3 specification of the Statement elements in the Package Metamodel

```
1  abstract class Statement extends NamedElement{
2      reference script: Script oppositeOf statements;
3      reference previous[0-1]  : Statement;
4      reference next[0-1] : Statement;
5  }
6
7  abstract class FileSystemStatement extends Statement{
8      reference files[1-*]: File;
9  }
10
11 abstract class EnvironmentStatement extends Statement{
12     reference environment: Environment;
13 }
14
15 abstract class PackageSettingStatement extends Statement {
16     reference pkgsetting : PackageSetting;
17 }
18
19 abstract class ControlStatement extends Statement {}
20
21 abstract class NeutralStatement extends Statement{}
22
23
24 class AdditionFileSytemStatement extends FileSystemStatement {}
25
26 class DeletionFileSystemStatement extends FileSystemStatement {}
27
28 class UpdateFileSytemStatement extends FileSystemStatement {}
29
```

```
30
31 class AdditionEnvironmentStatement extends EnvironmentStatement {}
32
33 class DeletionEnvironmentStatement extends EnvironmentStatement {}
34
35 class UpdateEnvironmentStatement extends EnvironmentStatement {}
36
37
38 class AdditionPackageSettingStatement extends PackageSettingStatement {}
39
40 class DeletionPackageSettingStatement extends PackageSettingStatement {}
41
42 class UpdatePackageSettingStatement extends PackageSettingStatement {}
```

A summarizing example is depicted in Fig. 5.7 which reports a fragment of the `postinst` script of the Debian Lenny `libapache2-mod-php5_5.2.6-5_amd64.deb` package. The code is represented in terms of a model (see the right-hand side of the figure) defined by means of the metaclasses presented above implemented on the Eclipse platform [BSM+03]. In particular, the copy operation of the file `php.ini-dist` represents a modification of the file system and is hence modeled as an `UpdateFileSystemStatement` element (see the arrow ⓒ). Once the `php5` module has been installed, the configuration of the `apache2` package has to be modified by enabling the new module. This operation is performed by executing the command `a2enmod` which is modeled as an `AdditionPackageSettingStatement` element (see the arrow ⓔ). Finally, the `UpdateEnvironmentStatement` element in the model (see the arrow ⓕ) represents the command which reloads the `Apache` Web server to update the environment with the previous modification.

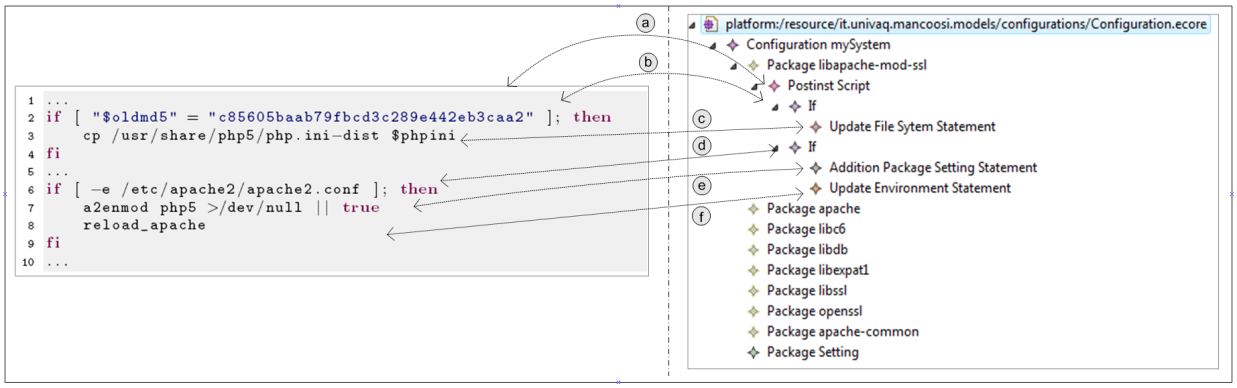

Figure 5.7: Fragment of the `libapache2-mod-php5_5.2.6-5_amd64.deb.postinst` script

The individual statements of a script may require an explicit control flow specifying their execution order. For instance, in the example depicted in Fig. 5.7 the statements previously discussed are executed only if some condition is met (see line 2 and line 6). In this respect, the abstract metaclass `ControlStatement` has been introduced with a number of specializations consisting of the following metaclasses

- `If`
- `Case`
- `Return`
- `Iterator`

which will be described in the rest of the section.

If metaclass

Conditional statements can be described by means of the `If` metaclass (see Listing 5.4) which is an extension of `ControlStatement`. The sample model depicted in Fig. 5.7 uses such a metaclass to represent the statements at lines `2-4` and lines `6-9`.

Listing 5.4: KM3 specification of the If Statement element in the Package Metamodel

```
1 class If extends ControlStatement {
2     reference condition container : BooleanExpression;
3     reference then [0-*] container : Statement;
4     reference else [0-*] container : Statement;
5 }
6
7 class BooleanExpression {
8     attribute value : String;
9 }
```

Case metaclass

Maintainer scripts might also use *case* statements in order to choose from a sequence of conditions, and execute a corresponding statement. In this respect the abstract metaclass `Case` has been introduced as an extension of `ControlStatement` (see lines `1-4` in Listing 5.5). The analysis described in the previous chapter has discovered that the case conditions which can be used depend on the type of the script in which it is specified (see Tab. 5.2.2)

Script type	Case conditions
postinst	configure, abort-upgrade, abort-remove, abort-deconfigure
postrm	purge, remove, upgrade, failed-upgrade, abort-install, abort-upgrade, disappear
prerm	remove, upgrade, deconfigure, failed-upgrade
preinst	install, upgrade, abort-upgrade

Table 5.1: Case conditions

In this respect, the metaclass `Case` is specialized in `CasePrerm`, `CasePostrm`, `CasePreinst`, `CasePostinst` (see lines `12-38`) in order to force the modeler to specify only the conditions that are admitted by the case statement being defined. For instance, `CasePrerm` elements can have conditions the value of which is restricted to the enumeration `CasePrermConditionValue` (see lines `52-57`).

Listing 5.5: KM3 specification of the Case Statement element in the Package Metamodel

```
1  abstract class Case extends ControlStatement {
2      attribute selector : String;
3      reference "default" container : DefaultCaseCondition;
4  }
5  abstract class Condition {
6      reference action container : Statement;
7  }
8
9  class DefaultCaseCondition extends Condition {
10 }
11
12 class CasePrerm extends Case {
13     reference conditions[0-*] container : CasePrermCondition;
14 }
15
16 class CasePrermCondition extends Condition {
17     attribute value  : CasePrermConditionValue;
18 }
```

```
19
20 class CasePostrm extends Case {
21     reference conditions[0-*] container : CasePostrmCondition;
22 }
23
24 class CasePostrmCondition extends Condition {
25     attribute value  : CasePostrmConditionValue;
26 }
27
28 class CasePreinst extends Case {
29     reference conditions[0-*] container : CasePreinstCondition;
30 }
31
32 class CasePreinstCondition extends Condition {
33     attribute value  : CasePreinstConditionValue;
34 }
35
36 class CasePostinst extends Case {
37     reference conditions[0-*] container : CasePostinstCondition;
38 }
39
40 class CasePostinstCondition extends Condition {
41     attribute value  : CasePostinstConditionValue;
42 }
43
44 class BasicCase extends ControlStatement {
45     reference conditions[0-*] container : BasicCaseCondition;
46 }
47
48 class BasicCaseCondition extends Condition{
49     attribute value  : String;
50 }
51
52 enumeration CasePrermConditionValue {
53     literal "remove";
54     literal "upgrade";
55     literal "deconfigure";
56     literal "failed-upgrade";
57 }
58
59 enumeration CasePostrmConditionValue {
60     literal "purge";
61     literal "remove";
62     literal "upgrade";
63     literal "failed";
64     literal "upgrade";
65     literal "abort-install";
66     literal "abort-upgrade";
67     literal "disappear";
68 }
69
70 enumeration CasePreinstConditionValue {
71     literal "install";
72     literal "upgrade";
73     literal "abort-upgrade";
74 }
75
76 enumeration CasePostinstConditionValue {
77     literal "configure";
78     literal "abort-upgrade";
79     literal "abort-remove";
80     literal "abort-deconfigure";
81 }
```

Iterator metaclass

To enable the specification of iterations on collection, the `Iterator` metaclass is defined with different specializations. In particular, scripts may have the necessity to iterate on the files contained in a given directory. In order to specify such a statement the metaclass `DirectoryIterator` is provided (see lines `4-6`). In order to specify iterations on the lines of a given file, the `FileContentIterator` is given. As specified in the metaclass `Script` in Listing 5.2, maintainer scripts may have input parameters which might be considered by some interations. In this respect, the `InputParameterIterator` metaclass is provided to iterate on the input parameters of a given script (see lines `12-14`).

The user might have the need for enumerations which can be specified as ordered sets of $(index, value)$ couples (see lines `24-31`). Once defined, enumerations can be iterated by means of `EnumerationIterator` instances (see lines `16-18`).

A number of scripts execute iterations on the characters of a string. In order to capture this case, the mataclass `StringIterator` at lines `20-22` has been introduced.

Listing 5.6: KM3 specification of the Iterator Statement elements in the Package Metamodel

```
1  abstract class Iterator extends ControlStatement {
2  }
3
4  class DirectoryIterator extends Iterator {
5      reference directory : File;
6  }
7
8  class FileContentIterator extends Iterator {
9      reference file : File;
10 }
11
12 class InputParameterIterator extends Iterator {
13     reference inputParameters[1-*] : InputParameter;
14 }
15
16 class EnumerationIterator extends Iterator {
17     reference "enumeration" : Enumeration;
18 }
19
20 class StringIterator extends Iterator {
21     reference string : StringEl;
22 }
23
24 class Enumeration {
25     reference elements[1-*] ordered : EnumerationElement;
26 }
27
28 class EnumerationElement {
29     attribute index : Integer;
30     attribute value : String;
31 }
```

Return metaclass

In order to specify the exit from a script the `Return` metaclass in Listing 5.7 is provided. The admitted values are given in the enumeration `ReturnStatementValue` which contains the literal `0` for specifying success, `1` in case of failures.

Listing 5.7: KM3 specification of the If Return element in the Package Metamodel

```
1  class Return extends ControlStatement {
2      attribute value : ReturnStatementValue;
```

```
}

enumeration ReturnStatementValue {
    literal "0";
    literal "1";
}
```

5.2.3 Template metaclasses

The analysis presented in the previous chapter has discovered that a large number of the maintainer scripts of the Debian Lenny and Fedora packages are generated by means of predefined templates which are reported in Appendix A.1 and Appendix A.2. This means that the Package metamodel has to provide the modeling constructs corresponding to those templates. The KM3 specification reported in Listing 5.8 specifies such metaclasses as an extension of the proper statements with respect to the modified configuration element. For instance, the *Postinst-Desktop* metaclass specified at lines `1,3` extends the `UpdateEnvironmentStatements` since it corresponds to the templates in Listing A.1 and Listing A.61 which update the system environment with the new MimeTypes embedded in the desktop files which have been installed. All the metaclasses which have been specified are summarized in Tab. 5.2.3: the column *Metaclass* contains the metaclasses which have been introduced to model the corresponding templates reported in the column *Represented templates*.

Listing 5.8: KM3 specification of the Debhelper templates in the Package Metamodel

```
-- Postinst-desktop
class PostinstDesktop extends UpdateEnvironmentStatement {
}

-- Postinst-doc-base
class PostinstDocBase extends AdditionEnvironmentStatement {
    reference document : File ;
}

-- Postinst-emacsen
class PostinstEmacsen extends UpdateEnvironmentStatement, UpdatePackageSettingStatement
    ↪{
    reference "package" : File;
}

-- Postinst-gconf
class PostinstGconf extends UpdatePackageSettingStatement {
    reference schemas[1-*]: File;

}

-- Postinst-gconf-defaults
class PostinstGconfDefaults extends UpdatePackageSettingStatement {
}

-- Postinst-icons
class PostinstIcons extends UpdateEnvironmentStatement {
    reference directories[1-*] : File;
}

-- Postinst-info
class PostinstInfo extends UpdateEnvironmentStatement {
    reference file : File ;
}

-- Postinst-init
class PostinstInit extends UpdateEnvironmentStatement  {
    reference scriptParam : Script;
    reference initParams [0-*] : Param;
```

Metaclass	Represented templates
Postinst-desktop	A.1, A.61
Postinst-doc-base	A.3
Postinst-emacsen	A.5
Postinst-gconf	A.9, A.54
Postinst-gconf-defaults	A.7
Postinst-icons	A.12, A.65
Postinst-info	A.14, A.57
Postinst-init	A.19, A.74
Postinst-init-nostart	A.16
Postinst-init-restart	A.17
Postinst-makeshlibs	A.22, A.70, A.72
Postinst-menu	A.26
Postinst-menu-method	A.24
Postinst-mime	A.39
Postinst-modules	A.28
Postinst-python	A.30
Postinst-scrollkeeper	A.32, A.59
Postinst-sgmlcatalog	A.34
Postinst-sharedmimeinfo	A.37, A.63
Postinst-suid	A.41
Postinst-udev	A.44
Postinst-usrlocal	A.45
Postinst-wm	A.48
Postinst-wm-noman	A.47
Postinst-xfonts	A.50, A.67
Postrm-debconf	A.52
Postrm-desktop	A.2, A.62
Postrm-gconf	A.11, A.56
Postrm-gconf-defaults	A.8
Postrm-icons	A.13
Postrm-init	A.21, A.76
Postrm-makeshlibs	A.23, A71, A.73
Postrm-menu	A.27
Postrm-menu-method	A.25
Postrm-mime	A.40, A.64
Postrm-modules	A.29
Postrm-scrollkeeper	A.33, A.60
Postrm-sgmlcatalog	A.36
Postrm-sharedmimeinfo	A.38, A.66
Postrm-suid	A.42
Postrm-xfonts	A.51, A.68
Preinst-udev	A.43
Preinst-user	A.69
Prerm-doc-base	A.4
Prerm-emacsen	A.6
Prerm-gconf	A.10, A.55, A.53
Prerm-info	A.15, A.58
Prerm-init	A.20, A.75
Prerm-init-norestart	A.18
Prerm-python	A.31
Prerm-sgmlcatalog	A.35
Prerm-usrlocal	A.46
Prerm-wm	A.49

Table 5.2: Metaclasses representing the recurrent templates

```
}

-- Postinst-init-nostart
class PostinstInitNostart extends UpdateEnvironmentStatement {
    reference scriptParam : Script;
    reference initParams[0-*] : Param;
    reference errorHandler : File;
}

-- Postinst-init-restart
class PostinstInitRestart extends UpdateEnvironmentStatement {
    reference scriptParam : Script;
    reference initParams[0-*] : Param;
    reference errorHandler : File;
}

-- Postinst-makeshlibs
class PostinstMakeshlibs extends UpdateEnvironmentStatement {
}

-- Postinst-menu
class PostinstMenu extends UpdateEnvironmentStatement {
}

-- Postinst-menu-method
class PostinstMenuMethod extends UpdateEnvironmentStatement {
    reference "package" : Package;
}

-- Postinst-mime
class PostinstMime extends UpdateEnvironmentStatement {
}

-- Postinst-modules
class PostinstModules extends UpdateEnvironmentStatement {
    reference kvers : StringParam;
}

-- Postinst-python
class PostinstPython extends UpdateEnvironmentStatement {
    reference pyver : IntParam;
    reference dirlist[1-*] : File;
}

-- Postinst-scrollkeeper
class PostinstScrollkeeper extends UpdateEnvironmentStatement {
}

-- Postinst-sgmlcatalog
class PostinstSgmlcatalog extends UpdateEnvironmentStatement {
    reference centralcat : File ;
    reference ordcats : File ;
}

-- Postinst-sharedmimeinfo
class PostinstSharedmimeinfo extends UpdateEnvironmentStatement {
}

-- Postinst-suid
class PostinstSuid extends  UpdateEnvironmentStatement, UpdateFileSytemStatement {
    reference "package" : Package;
    reference file : File;
    reference owner : StringParam;
    reference group : StringParam;
    reference perms : StringParam;
}

-- Postinst-udev
class PostinstUdev extends UpdateFileSytemStatement {
    reference old : File;
```

```
109     reference rule : File;
110 }
111
112 -- Postinst-usrlocal
113 class PostinstUsrlocal extends UpdateFileSytemStatement {
114     reference  dir : StringParam;
115     reference  mode: StringParam;
116     reference  user: StringParam;
117     reference  group : StringParam;
118 }
119
120 -- Postinst-wm
121 class PostinstWm extends UpdateEnvironmentStatement {
122     reference wm : StringParam;
123     reference wmman : StringParam;
124     reference priority : StringParam;
125 }
126
127 -- Postinst-wm-noman
128 class PostinstWmNoman extends UpdateEnvironmentStatement {
129     reference wm : StringParam;
130     reference priority : StringParam;
131 }
132
133 -- Postinst-xfonts
134 class PostinstXfonts extends UpdateEnvironmentStatement {
135     reference cmds[0-*] : Statement;
136 }
137
138 -- Postrm-debconf
139 class PostrmDebconf extends UpdateEnvironmentStatement{
140 }
141
142 -- Postrm-desktop
143 class PostrmDesktop extends UpdateEnvironmentStatement {
144 }
145
146 -- Postrm-gconf
147 class PostrmGconf extends DeletionFileSystemStatement {
148     reference schemas[1-*] : File;
149 }
150
151 -- Postrm-gconf-defaults
152 class PostrmGconfDefaults extends UpdateEnvironmentStatement {
153 }
154
155 -- Postrm-icons
156 class PostrmIcons extends UpdateEnvironmentStatement {
157     reference dirs[1-*] : File;
158 }
159
160 -- Postrm-init
161 class PostrmInit extends UpdateEnvironmentStatement, DeletionFileSystemStatement {
162     reference scriptParam : Script;
163 }
164
165 -- Postrm-makeshlibs
166 class PostrmMakeshlibs extends UpdateEnvironmentStatement {
167 }
168
169 -- Postrm-menu
170 class PostrmMenu extends UpdateEnvironmentStatement {
171 }
172
173 -- Postrm-menu-method
174 class PostrmMenuMethod extends UpdateEnvironmentStatement, UpdateFileSytemStatement {
175     reference "package" : Package;
176 }
177
178 -- Postrm-mime
```

```
179 class PostrmMime extends UpdateEnvironmentStatement {
180 }
181
182 -- Postrm-modules
183 class PostrmModules extends UpdateEnvironmentStatement {
184     reference kvers : IntParam;
185 }
186
187 -- Postrm-scrollkeeper
188 class PostrmScrollkeeper extends UpdateEnvironmentStatement {
189 }
190
191 -- Postrm-sgmlcatalog
192 class PostrmSgmlcatalog extends UpdateFileSytemStatement {
193     reference centralcat : File;
194 }
195
196 -- Postrm-sharedmimeinfo
197 class PostrmSharedmimeinfo extends UpdateEnvironmentStatement {
198 }
199
200 -- Postrm-suid
201 class PostrmSuid extends UpdateEnvironmentStatement {
202     reference "package" : Package;
203     reference file : File;
204 }
205
206 -- Postrm-xfonts
207 class PostrmXfonts extends  UpdateEnvironmentStatement {
208     reference cmds[0-*] : Statement;
209 }
210
211 -- Preinst-udev
212 class PreinstUdev extends DeletionFileSystemStatement {
213     reference old : File;
214     reference rule : File;
215     reference "package" : Package;
216 }
217
218 -- Preinst-user
219 class PreinstUser extends UpdateFileSytemStatement {
220     reference  dir : StringParam;
221     reference  mode: StringParam;
222     reference  user: StringParam;
223     reference  group : StringParam;
224 }
225
226 -- Prerm-doc-base
227 class PrermDocBase extends UpdateEnvironmentStatement {
228     reference doc: File;
229 }
230
231 -- Prerm-emacsen
232 class PrermEmacsen extends UpdateEnvironmentStatement, UpdatePackageSettingStatement {
233     reference "package" : Package;
234 }
235
236 -- Prerm-gconf
237 class PrermGconf extends UpdatePackageSettingStatement {
238     reference schemas[0-*] : File;
239 }
240
241 -- Prerm-info
242 class PrermInfo extends UpdateEnvironmentStatement, DeletionPackageSettingStatement {
243     reference file : File;
244 }
245
246 -- Prerm-init
247 class PrermInit extends UpdateEnvironmentStatement {
248     reference scriptParam : File;
```

```
249 }
250
251 -- Prerm-init-norestart
252 class PrermInitNorestart extends UpdateEnvironmentStatement {
253     reference scriptParam : File;
254 }
255
256 -- Prerm-python
257 class PrermPython extends DeletionFileSystemStatement{
258     reference "package" : Package;
259 }
260
261 -- Prerm-sgmlcatalog
262 class PrermSgmlcatalog extends UpdateEnvironmentStatement,
        ↪DeletionPackageSettingStatement {
263     reference centralcat : File;
264 }
265
266 -- Prerm-usrlocal
267 class PrermUsrlocal extends DeletionFileSystemStatement {
268 }
269
270 -- Prerm-wm
271 class PrermWm extends DeletionPackageSettingStatement {
272     reference wm : StringParam;
273 }
274
275 class Param {
276     attribute value : String;
277 }
278
279 class StringParam extends Param {
280 }
281
282 class IntParam extends Param{ }
```

5.3 Log metamodel

The metamodel depicted in Listing. 5.9 is a step towards the development of a transactional model of upgradeability that will allow us to roll-back long upgrade history, restoring previous configurations. In particular, the metaclass `Transaction` (see lines `5-10`) refers to the set of statements which have been executed from a source configuration leading to a target one. For instance, according to the sample log model in Fig. 5.9, the installation of the package `libapache-mod-php5` modifies the file system (see the statement `afss1` which represents the addition of the file `f1`) and updates the Apache configuration (see the element `upss1`).

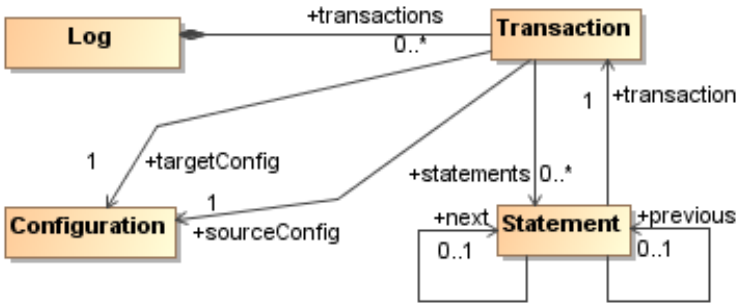

Figure 5.8: Fragment of the Log metamodel

Listing 5.9: KM3 specification of the Log metamodel

```
1  class Log extends NamedElement{
2        reference transactions[0-*] container: Transaction oppositeOf log;
3    }
4
5    class Transaction extends NamedElement{
6        reference statements[0-*] : Statement oppositeOf trans;
7        reference sourceConfig : Configuration ;
8        reference targetConfig : Configuration ;
9        reference log: Log oppositeOf transactions;
10   }
```

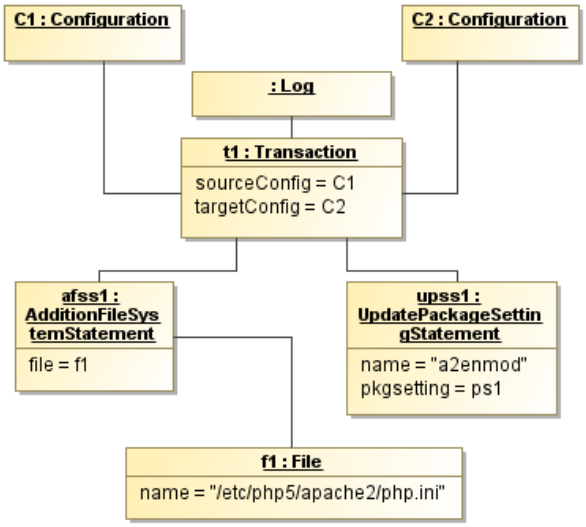

Figure 5.9: Sample Log model

The usefulness of log models like the one in Fig. 5.9 is manyfold and accounts for several roll-back needs:

(a) *Preference roll-back:* the user wants to recover a previous configuration, for whatever reason. For instance, the user is not in need of PHP5 anymore and wants to remove the installed package `libapache-mod-php5`. In this case, the configuration `C1` can be recovered by executing the dual operation of each statement in the transaction between `C1` and `C2`. Note that the log models have all the information necessary to roll-back to any previous valid configuration not necessarily a contiguous one;

(b) *Compensate model incompleteness*: as already discussed, upgrade simulation is not complete with respect to upgrades, and undetected failures can be encountered while deploying upgrades on the real system. For instance, the addition of the file `php.ini` during the installation of the package `libapache-mod-php5` can raise faults because of disk errors. In this case we can exploit the information stored in the log model to retrieve the fallacious statements and to roll-back to the configuration from which the broken transaction has started.

(c) *"Live" failures:* the proposed approach does not mandate to pre-simulate upgrades. In fact, it is possible as well to avoid simulation and have metamodeling supervise upgrades to detect invalid configurations as soon as they are reached. At that point, if any, log models come into play and enable rolling back deployed changes to bring the system back to a previous valid configuration.

Chapter 6

Supporting the evolution of the MANCOOSI metamodels

Evolution is an inevitable aspect which affects the whole life-cycle of software systems [LB85]. In general, artefacts can be subject to many kinds of changes, which range from requirements through architecture and design, to source code, documentation and test suites. Similarly to other software artefacts, metamodels can evolve over time too [Fav03]. Accordingly, models need to be *co-adapted* in order to remain compliant to the metamodel and not become eventually invalid. When manually operated the adaptation is error-prone and can give place to inconsistencies between the metamodel and the related artefacts. Such an issue becomes very relevant when dealing with FOSS systems whose specifications consist of large models and even small metamodel modifications require adaptation steps to be performed on all the already existing models.

In this chapter we outline a transformational approach to model co-evolution we have proposed in [CDEP08]. By means of the introduced techniques, well-defined adaptation steps are generated directly from the modifications the metamodel underwent. The approach is based on a model difference representation [CDP07] which is used to specify in a *difference* model the metamodel changes. Thus, the co-adaptation is given as a higher-order model transformation which takes the difference model recording the metamodel evolution and generates a model transformation able to produce the co-evolution of models.

The overall chapter outlines the co-evolution approach which will be used throughout the duration of the project in case the metamodels described in the previous chapter have to be refined and the already existing models have to be consistently adapted.

The chapter is structured as follows: Section 6.1 presents the problem of model co-evolution by means of a running example which will be used throughout the chapter. Section 6.2 outlines the techniques which are used to represent the modifications that distinguish two versions of a same metamodel. The transformational approach to model co-evolution relies on such techniques as discussed in Section 6.3.

6.1 Metamodel evolution and model co-evolution

Metamodels are expected to evolve during their life-cycle, thus causing possible problems to existing models which conform to the old version of the metamodel and do not conform to the

new version anymore. The problem is due to the incompatibility between the metamodel revisions and a possible solution is the adoption of mechanisms of model co-evolution, i.e., models need to be migrated in new instances according to the changes of the corresponding metamodel. Unfortunately, model co-evolution is not always simple and presents intrinsic difficulties which are related to the kind of evolution the metamodel has been subject to. Going into more details, metamodels may evolve in different ways: some changes may be additive and independent from the other elements, thus requiring no or little instance revision. In other cases metamodel manipulations introduce incompatibilities and inconsistencies which cannot be easily (and automatically) resolved. For instance, additions or refinements to the metamodels presented in the previous chapter require the adaptation of all the system configuration and package models to re-establish their conformance to the new metamodels.

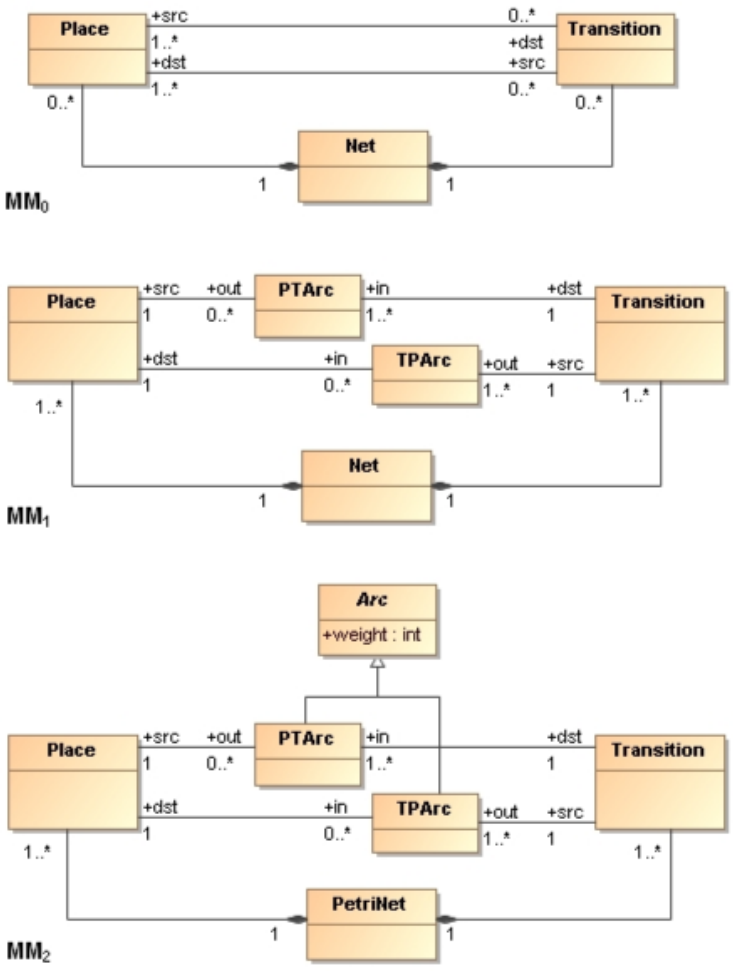

Figure 6.1: Petri Net metamodel evolution

To simplify the presentation of the problems related to the co-evolution task the sample Petri Net metamodel evolution depicted in Fig. 6.1 will be considered in the rest of the chapter. The initial Petri Net ($\texttt{MM}_0$) consists of `Places` and `Transitions`; moreover, places can have source and/or destination transitions, whereas transitions must link source and destination places (`src` and `dst` association roles, respectively). In the new metamodel $\texttt{MM}_1$, each `Net` has at least one `Place` and one `Transition`. Besides, arcs between places and transitions are made explicit by extracting `PTArc` and `TPArc` metaclasses. This refinement allows to add further properties

to relationships between places and transitions. For example, the Petri Net formalism can be extended by annotating arcs with weights. As `PTArc` and `TPArc` both represent arcs, they can be generalized by a superclass, and a new integer metaproperty can be added in it. Therefore, an abstract class `Arc` encompassing the integer metaproperty `weight` has been added in $\texttt{MM}_2$ revision of the metamodel. Finally, `Net` has been renamed into `PetriNet`.

The revisions illustrated so far can invalidate existing instances; therefore, each version needs to be analyzed to comprehend the various kind of updates it has been subject to and, eventually, to elicit the necessary adaptations of corresponding models. Metamodel manipulations can be classified by their corrupting or non-corrupting effects on existing instances [GKP07]:

- *non-breaking changes*: changes which do not break the conformance of models to the corresponding metamodel;
- *breaking and resolvable changes*: changes which break the conformance of models even though they can be automatically co-adapted;
- *breaking and unresolvable changes*: changes which break the conformance of models which cannot automatically co-evolved, in which case user intervention is required.

In other words, *non-breaking changes* consist of additions of new elements in a metamodel $\texttt{MM}$ leading to $\texttt{MM}'$ without compromising models which conform to $\texttt{MM}$ and thus, in turn, conform to $\texttt{MM}'$. For instance, in the metamodel $\texttt{MM}_2$ illustrated in Fig. 6.1 the abstract metaclass `Arc` has been added as a generalization of the `PTArc` and `TPArc` metaclasses (without considering the new attribute `weight`). After such a modification, models conforming to $\texttt{MM}_1$ still conform to $\texttt{MM}_2$ and co-evolution is not necessary. Unfortunately, this is not always the case since in general changes may break models even though sometimes automatic resolution can be performed, i.e., when facing *breaking and resolvable changes*. For instance, the Petri Net metamodel $\texttt{MM}_1$ in Fig. 6.1 is enriched with the new `PTArc` and `TPArc` metaclasses. Such a modification breaks the models that conform to $\texttt{MM}_0$ since, according to the new metamodel $\texttt{MM}_1$, `Place` and `Transition` instances can not be directly related, but `PTArc` and `TPArc` elements are required. However, models can be automatically migrated by adding for each couple of `Place` and `Transition` entities two additional `PTArc` and `TPArc` instances between them. For instance, the sample Petri Net model depicted in Fig. 6.2.a and which conforms to $\texttt{MM}_0$ can be automatically adapted leading to the model in Fig. 6.2.b.

Manual interventions are often needed to solve breaking changes like, for instance, the addition of the new attribute `weight` to the class `Arc` of $\texttt{MM}_2$ in Fig. 6.1 which were not specified in $\texttt{MM}_1$.

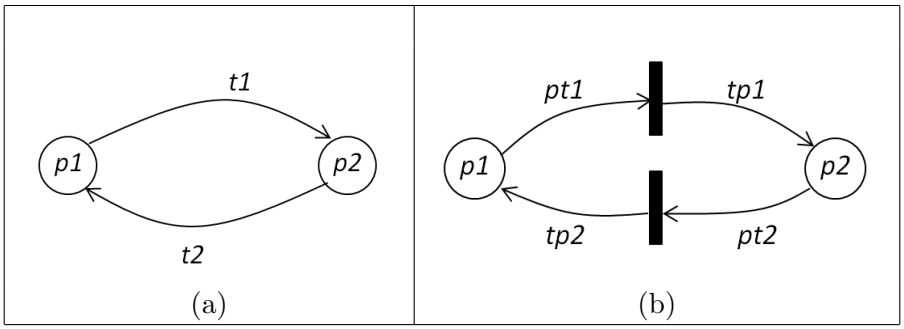

Figure 6.2: Sample Petri Net model adaptation

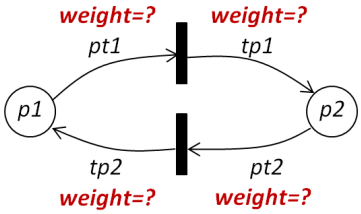

Figure 6.3: Sample Petri Net model which requires human intervention

The models conforming to $\texttt{MM}_1$ cannot be automatically co-evolved since only a human intervention can introduce the missing information related to the weight of the arc being specified, or otherwise default values have to be considered (see Fig. 6.3). We refer to such situations as *breaking and unresolvable changes*.

All the scenarios of model co-adaptations can be managed with respect to the possible metamodel modifications which can be distinguished into *additive*, *subtractive*, and *updative*. In particular, with additive changes we refer to metamodel element additions which in turn can be further distinguished as follows:

- *Add metaclass*: introducing new metaclasses is a common practice in metamodel evolution which gives place to metamodel extensions. Adding new metaclasses raises co-evolution issues only if the new elements are mandatory with respect to the specified cardinality. In this case, new instances of the added metaclass have to be accordingly introduced in the existing models;
- *Add metaproperty*: this is similar to the previous case since a new metaproperty may be obligatory or not with respect to the specified cardinality. The existing models maintain the conformance to the considered metamodel if the addition occurs in abstract metaclasses without subclasses; in other cases, human intervention is required to specify the value of the added property in all the involved model elements;
- *Generalize metaproperty*: a metaproperty is generalized when its multiplicity or type are relaxed. For instance, if the cardinality `3..n` of a sample metaclass `MC` is modified in `0..n`, no co-evolution actions are required on the corresponding models since the existing instances of `MC` still conform to the new version of the metaclass;
- *Pull metaproperty*: a metaproperty `p` is pulled in a superclass `A` and the old one is removed from a subclass `B`. As a consequence, the instances of the metaclass `A` have to be modified by inheriting the value of `p` from the instances of the metaclass `B`;
- *Extract superclass*: a superclass is extracted in a hierarchy and a set of properties is pulled on. If the superclass is abstract model instances are preserved, otherwise the effects are referable to metaproperty pulls.

Subtractive changes consist of the deletion of some of the existing metamodel elements as described in the following:

- *Eliminate metaclass*: a metaclass is deleted by giving place to a sub metamodel of the initial one. In general, such a change induces in the corresponding models the deletions of all the metaclass instances. Moreover, if the involved metaclass has subclasses or it is referred by other metaclasses, the elimitation also causes side effects to the related entities;

Change type	Change
Non-breaking changes	Generalize metaproperty Add (non-obligatory) metaclass Add (non-obligatory) metaproperty
Breaking and resolvable changes	Extract (abstract) superclass Eliminate metaclass Eliminate metaproperty Push metaproperty Flatten hierarchy Rename metaelement Move metaproperty Extract/inline metaclass
Breaking and unresolvable changes	Add obligatory metaclass Add obligatory metaproperty Pull metaproperty Restrict metaproperty Extract (non-abstract) superclass

Table 6.1: Changes classification

- *Eliminate metaproperty*: a property is eliminated from a metaclass, it has the same effect of the previous modification;
- *Push metaproperty*: pushing a property in subclasses means that it is deleted from an initial superclass `A` and then cloned in all the subclasses `C` of `A`. If `A` is abstract then such a metamodel modification does not require any model co-adaptation, otherwise all the instances of `A` and its subclasses need to be accordingly modified;
- *Flatten hierarchy*: to flatten a hierarchy means eliminating a superclass and introducing all its properties into the subclasses. This scenario can be referred to metaproperty pushes;
- *Restrict metaproperty*: a metaproperty is restricted when its multiplicity or type are enforced. It is a complex case where instances need to be co-adapted or restricted. Restricting the upper bound of the multiplicity requires a selection of certain values to be deleted. Increasing the lower bound requires new values to be added for the involved element which usually are manually provided. Restricting the type of a property requires type conversion for each value.

Finally, a new version of the model can consist of some updates of already existing elements leading to updative modifications which can be grouped as follows:

- *Rename metaelement*: renaming is a simple case in which the change needs to be propagated to existing instances and can be performed in an automatic way;
- *Move metaproperty*: it consists of moving a property `p` from a metaclass `A` to a metaclass `B`. This is a resolvable change and the existing models can be easily co-evolved by moving the property `p` from all the instances of the metaclass `A` to the instances of `B`;
- *Extract/inline metaclass*: extracting a metaclass means to create a new class and move the relevant fields from the old class into the new one. Vice versa, to inline a metaclass means to move all its features into another class and delete the former. Both metamodel refactorings induce automated model co-evolutions.

The classification illustrated so far is summarized in Tab. 6.1 and makes evident the fundamental role of evolution representation. At a first glance, it seems that the classification does not

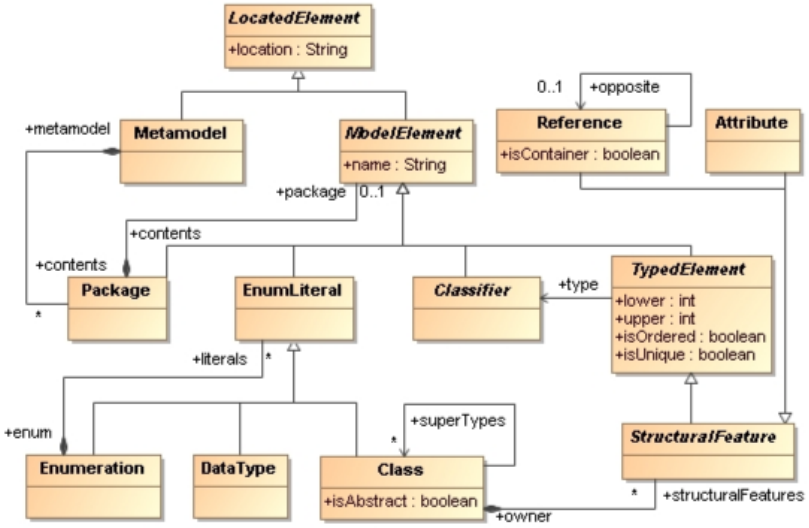

Figure 6.4: KM3 metamodel

encompass *references* that are associations amongst metaclasses. However, references can be considered properties of metaclasses at the same level of attributes.

Metamodel evolutions can be precisely categorized by understanding the kind of modifications a metamodel undergone. Moreover, starting from the classification it is possible to adopt adequate countermeasures to co-evolve existing instances. Nonetheless, it is worth noting that the classification summarized in Tab. 6.1 is based on a clear distinction between the metamodel evolution categories. Unfortunately, in real world experiences the evolution of a metamodel can not be reduced to a sequence of atomic changes, generally several types of changes are operated as affecting multiple elements with different impacts on the co-adaptation. Furthermore, the entities involved in the evolution can be related one another. Therefore, since co-adaptation mechanisms are based on the described change classification, a metamodel adaptation will need to be decomposed in terms of the induced co-evolution categories. The possibility to have a set of dependences among the several parts of the evolution makes the updates not always distinguishable as single atomic steps of the metamodel revision, but requires a further refinement of the classification as introduced in the next section and discussed in details in Sect. 6.3.

6.2 Metamodel difference representation

The problem of model differences is intrinsically complex and requires specialized algorithms and notations to match the abstraction level of models [LZG04]. Recently, in [CDP07, RV08] two similar techniques have been introduced to represent differences as models, hereafter called *difference models*; interestingly these proposals combine the advantages of declarative difference representations and enable the reconstruction of the final model by means of automated transformations which are inherently defined in the approaches. In the rest of the section, we recall the difference representation approach defined in [CDP07] in order to provide the reader with the technical details which underpin the solution proposed in Sect. 6.3.

Although the work in [CDP07] has been introduced to deal with model revisions, it is also easily adaptable to metamodel evolutions. In fact, a metamodel is a model itself, which conforms to a metamodel referred to as the meta metamodel [BÓ5]. For presentation purposes, the

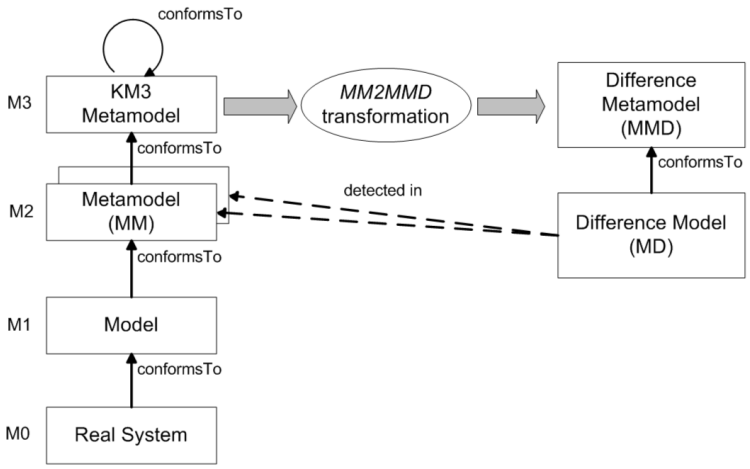

Figure 6.5: Overall structure of the model difference representation approach

KM3 language in Fig. 6.4 is considered, even though the solution can be generalized to any metamodeling language like OMG/MOF [Obj03b] or EMF/Ecore [BSM+03].

The overall structure of the change representation mechanism is depicted in Fig. 6.5: given two *base metamodels* $\texttt{MM}_1$ and $\texttt{MM}_2$ which conform to an arbitrary *base meta metamodel* (KM3 in our case), their difference conforms to a *difference metamodel* `MMD` derived from KM3 by means of an automated transformation `MM2MMD`. The base meta metamodel, extended as prescribed by such a transformation, consists of new constructs able to represent the possible modifications that can occur on metamodels and which can be grouped as follows:

- *additions*: new elements are added in the initial metamodel; with respect to the classification given in Sect. 6.1, *Add metaclass* and *Extract superclass* involve this kind of change;
- *deletions*: some of the existing elements are deleted as a whole. *Eliminate metaclass* and *Flatten hierarchy* fall in this category of manipulations;
- *changes*: a new version of the metamodel being considered can consist of updates of already existing elements. For instance, *Rename metaelement* and *Restrict metaproperty* require this type of modification. The addition and deletion of metaproperty (i.e., *Add metaproperty* and *Eliminate metaproperty*, respectively) are also modelled through this construct. In fact, when a metaelement is included in a container the manipulation is represented as a *change* of the container itself.

In order to represent the differences between the Petri Net metamodel revisions, the extended KM3 meta metamodel depicted in Fig. 6.6 is generated by applying the `MM2MMD` transformation in Fig. 6.5 previously mentioned. For each metaclass `MC` of the KM3 metamodel, the additional metaclasses `AddedMC`, `DeletedMC`, and `ChangedMC` are generated. For instance, the metaclass `Class` in Fig. 6.4 induces the generation of the metaclasses `AddedClass`, `DeletedClass`, and `ChangedClass` as depicted in Fig. 6.6. In the same way, `Reference` metaclass induces `AddedReference`, `DeletedReference`, and `ChangedReference`.

The generated difference metamodel is able to represent all the differences amongst metamodels which conform to KM3. For instance, the model in Fig. 6.7 conforms to the generated metamodel in Fig. 6.6 and represents the differences between the Petri Net metamodels specified in Fig. 6.1. The differences depicted in such a model can be summarized as follows:

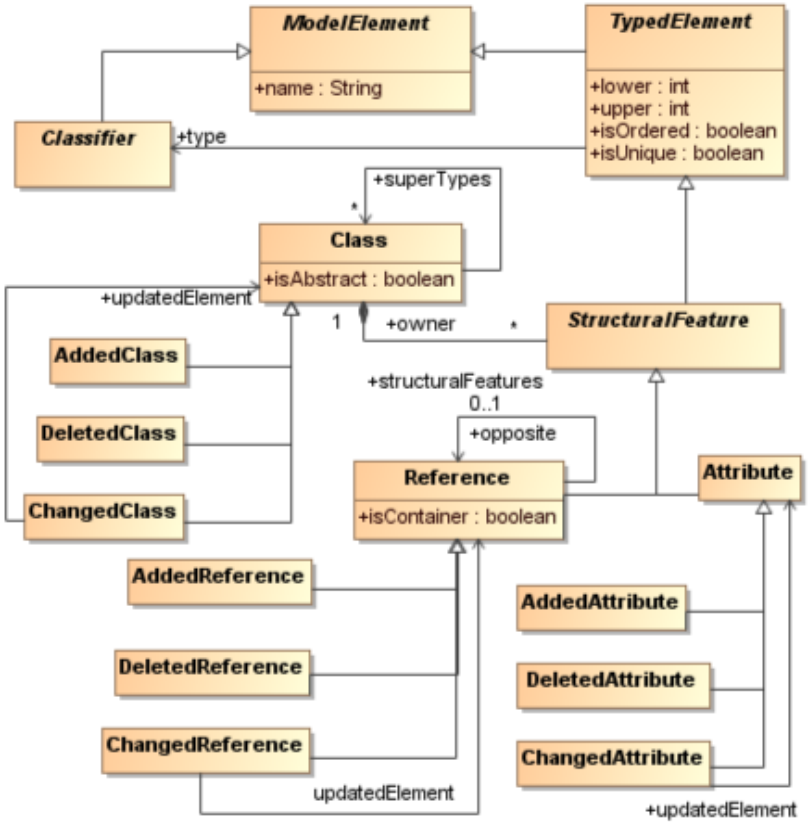

Figure 6.6: Generated difference KM3 metamodel

1) the addition of the new class `PTArc` in the MM_1 revision of the Petri Net metamodel is represented by means of an `AddedClass` instance, as illustrated by model difference $\Delta_{0,1}$ in Fig. 6.7. Moreover, the reference between `Place` and `Transition` named `dst` has been updated to link `PTArc` with name `out`. Analogously, the reverse reference named `src` has been manipulated to point `PTArc` and named as `in`. Two new references have been added through the corresponding `AddedReference` instances to realize the reverse links from `PTArc` to `Place` and `Transition`, respectively. Finally, the composition relationship between `Net` and `Place` has been updated by prescribing the existence of at least one `Place` through the `lower` property which has been updated from `0` to `1`. The same enforcement has been done to the composition between `Net` and `Transition`;

2) the addition of the new abstract class `Arc` in MM_2, together with its attribute `weight`, is represented through an instance of the `AddedClass` and the `AddedAttribute` metaclasses in the $\Delta_{1,2}$ delta of Fig. 6.7. In the meanwhile, `PTArc` and `TPArc` classes are made specializations of `Arc`. Finally, `Net` entity is renamed as `PetriNet`.

Difference models like the one in Fig. 6.7 can be obtained by using today's available tools like EMFCompare [Tou] and SiDiff [TBWK07], which are not discussed here.

The representation mechanism used so far allows to identify changes which occurred in a metamodel revision and satisfies a number of properties, as illustrated in [CDP07]. One of them is the *compositionality*, i.e., the possibility to combine difference models in interesting constructions like the sequential and the parallel compositions, which in turn result in valid difference

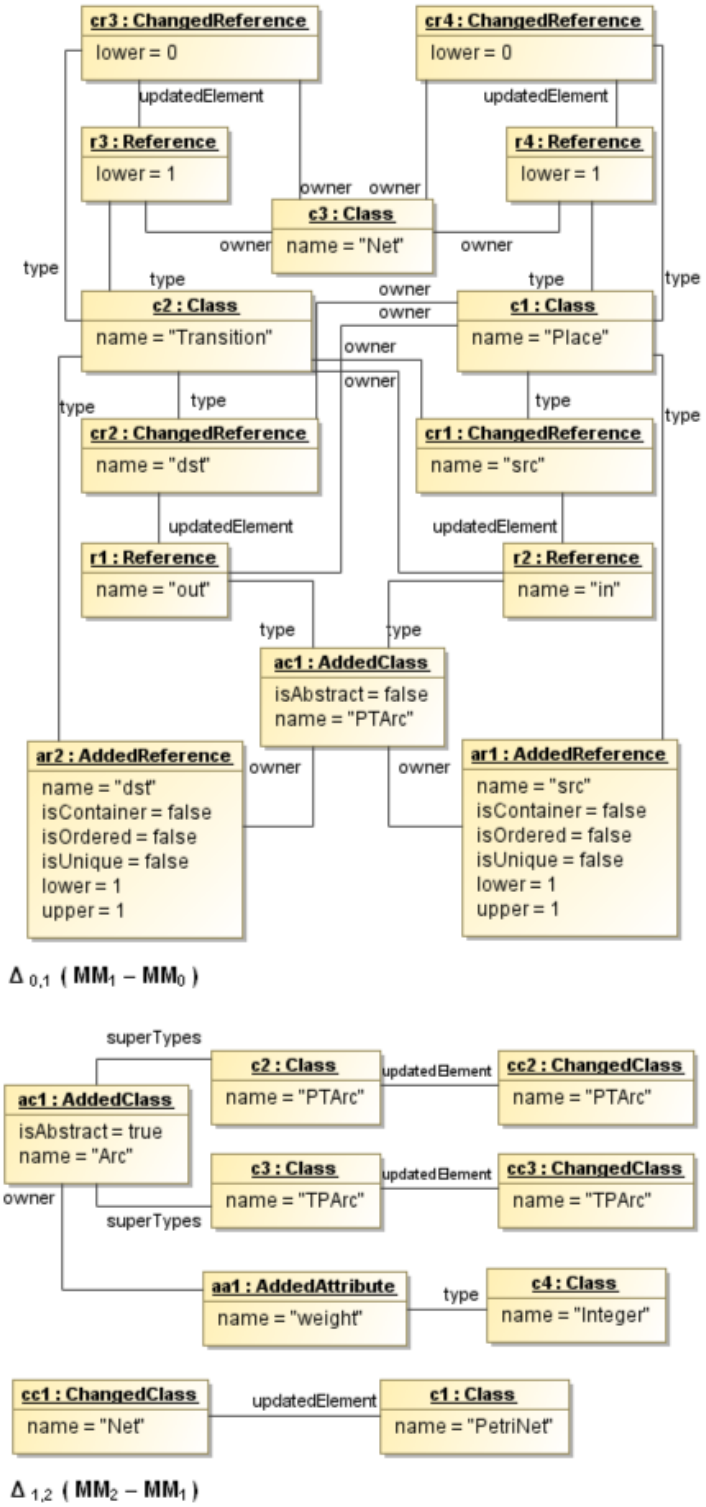

Figure 6.7: Subsequent Petri Net metamodel adaptations

models themselves. For the sake of simplicity, let us consider only two modifications over the initial model: the sequential composition of such manipulations corresponds to merging the modifications conveyed by the first document and then, in turn, by the second one in a resulting difference model containing a minimal difference set, i.e., only those modifications which have not been overridden by subsequent manipulations. Whereas, parallel compositions are exploited to combine modifications operated from the same ancestor in a concurrent way. In case both manipulations are not affecting the same elements they are said *parallel independent* and their composition is obtained by merging the difference models by interleaving the single changes and assimilating it to the sequential composition. Otherwise, they are referred to as *parallel dependent* and conflict issues can arise which need to be detected and resolved [Cic08].

Finally, difference documentation can be exploited to re-apply changes to arbitrary input models (see [CDP07] for further details) and for managing model co-evolution induced by metamodel manipulations. In the latter case, once differences between metamodel versions have been detected and represented, they have to be partitioned in resolvable and non resolvable scenarios in order to adopt the corresponding resolution strategy. However, this distinction is not always feasible because of parallel dependent changes, i.e., situations where multiple changes are mixed and interdependent on one another, like when a resolvable change is in some way related with a non-resolvable one, for instance. In those cases, deltas have to be decomposed in order to isolate the non-resolvable portion from the resolvable one, as illustrated in the next section.

6.3 Transformational adaptation of models

This section proposes a transformational approach able to consistently adapt existing models with respect to the modifications occurred in the corresponding metamodels. The proposal is based on model transformation and the difference representation techniques presented in the previous section. In particular, given two versions $\texttt{MM}_1$ and $\texttt{MM}_2$ of the same metamodel (see Fig. 6.8.a), their differences are recorded in a difference model Δ, whose metamodel `KM3Diff` is automatically derived from `KM3` as described in Sect. 6.2. In realistic cases, the modifications consist of an arbitrary combination of the atomic changes summarized in Tab. 6.1. Hence, a difference model formalizes all kind of modifications, i.e., non-breaking, breaking resolvable and unresolvable ones. This poses additional difficulties since current approaches (e.g. [Wac07, GKP07]) do not provide any support to co-adaptation when the modifications are given without explicitly distinguishing among breaking resolvable and unresolvable changes. Our approach consists of the following steps:

i) automatic decomposition of Δ in two disjoint (sub) models, Δ_R and $\Delta_{\neg R}$, which denote breaking resolvable and unresolvable changes;

ii) if Δ_R and $\Delta_{\neg R}$ are *parallel independent* (see previous section) then we separately generate the corresponding co-evolutions;

iii) if Δ_R and $\Delta_{\neg R}$ are *parallel dependent*, they are further refined to identify and isolate the interdependencies causing the interferences.

The distinction between *ii)* and *iii)* is due to the fact that when two modifications are not independent their effects depend on the order in which the changes occur leading to non confluent situations. The confluence can still be obtained by removing those modifications which caused the conflicts as described in Sect. 6.3.2.

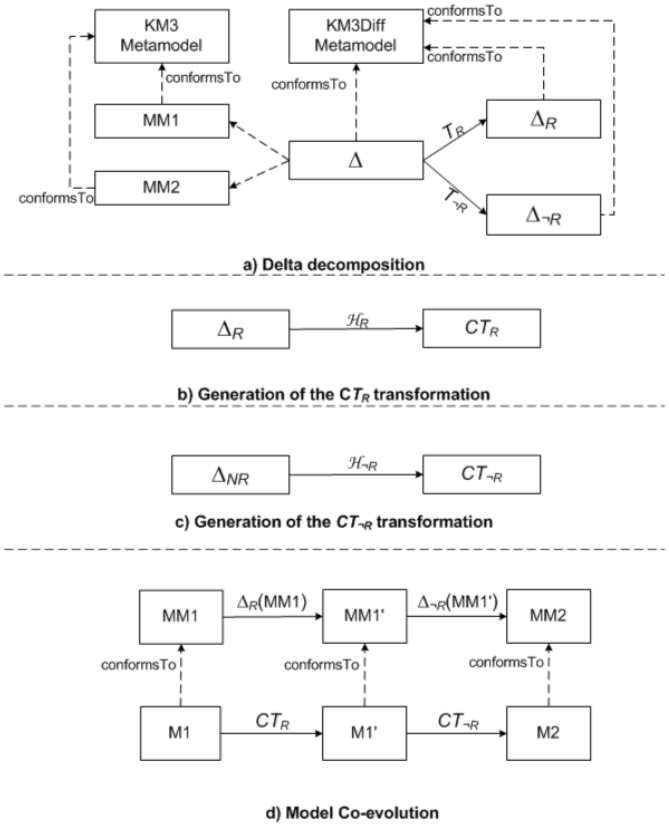

Figure 6.8: Overall co-evolution approach

The general approach is outlined in Fig. 6.8 where dotted and solid arrows represent conformance and transformation relations, respectively, and square boxes are any kind of models, i.e., models, difference models, metamodels, and even transformations. In particular, the decomposition of Δ is given by two model transformations, T_R and $T_{\neg R}$ (right-hand side of Fig. 6.8.a). Co-evolution actions are directly obtained as model transformations from metamodel changes by means of higher-order transformations, i.e., transformations which produce other transformations [BÓ5]. More specifically, the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ (see Fig. 6.8.b and 6.8.c) take Δ_R and $\Delta_{\neg R}$ and produce the (co-evolving) model transformations CT_R and $CT_{\neg R}$, respectively. Since Δ_R and $\Delta_{\neg R}$ are parallel independent CT_R and $CT_{\neg R}$ can be applied in any order because they operate to disjoint sets of model elements, or in other words

$$(CT_{\neg R} \cdot CT_R)(M_1) = (CT_R \cdot CT_{\neg R})(M_1) = M_2$$

with M_1 and M_2 models conforming to the metamodel $\texttt{MM}_1$ and $\texttt{MM}_2$, respectively (see Fig. 6.8.d).

The next sections illustrate the approach and its implementation. In particular, we describe the decomposition of Δ and the generation of the co-evolving model-transformations for the case of parallel independent breaking resolvable and unresolvable changes. Finally, in Sect. 6.3.2 we outline how to remove interdependencies from parallel dependent changes in order to generalize the solution provided in Sect. 6.3.1.

6.3.1 Parallel independent changes

The generation of the co-evolving model transformations is described in the rest of the section by means of the evolutions the `PetriNet` metamodel has been subject to in Fig. 6.1. The differences between the subsequent metamodel versions are given in Fig. 6.7 and have, in turn, to be decomposed to distinguish breaking resolvable and unresolvable modifications.

In particular, the difference $\Delta_{(0,1)}$ from $\texttt{MM}_0$ to $\texttt{MM}_1$ consists of two atomic modifications, i.e., an *Extract metaclass* and a *Restrict metaproperty* change (according to the classification in Tab. 6.1), which are referring to different sets of model elements. The approach is able to detect parallel independence by verifying that the eventual decomposed differences have an empty intersection. Since *a)* the previous atomic changes are breaking resolvable and unresolvable, and *b)* they do not share any model element, then $\Delta_{(0,1)}$ is decomposed by T_R and $T_{\neg R}$ into the parallel independent $\Delta_{R(0,1)}$ and $\Delta_{\neg R(0,1)}$, respectively. In fact, the former contains the extract metaclass action which affects the elements `Place` and `Transition`, whereas the latter holds the restrict metaproperty changes consisting of the reference modifications in the metaclass `Net`. Analogously, the same decomposition can be operated on $\Delta_{(1,2)}$ (denoting the evolution from $\texttt{MM}_1$ to $\texttt{MM}_2$) to obtain $\Delta_{R(1,2)}$ and $\Delta_{\neg R(1,2)}$ since the denoted modifications do not conflict one another. In fact, the *Rename metaelement* change (represented by `cc1` and `c1` in Fig. 6.7.b) is applied to `Net`, whereas the *Add obligatory metaproperty* operation involves the new metaclass `Arc` which is supertype of the `PTArc` and `TPArc` metaclasses.

As previously said, once the Δ is decomposed the higher-order transformations $\mathcal{H}_R$ and $\mathcal{H}_{\neg R}$ detect the occurred metamodel changes and accordingly generate the co-evolution to adapt the corresponding models. In the current implementation, model transformations are given in ATL, a QVT compliant language part of the AMMA platform [BJRV04] which contains a mixture of declarative and imperative constructs. A fragment of the $\mathcal{H}_R$ transformation is reported In the Listing 6.1: it consists of a module specification containing a header section (lines 1-2), transformation rules (lines 4-41) and a number of helpers which are used to navigate models and to define complex calculations on them. In particular, the header specifies the source models, the corresponding metamodels, and the target ones. Since the $\mathcal{H}_R$ transformation is higher-order, the target model conforms to the ATL metamodel which essentially specifies the abstract syntax of the transformation language. Moreover, $\mathcal{H}_R$ takes as input the model which represents the metamodel differences conforming to `KM3Diff`.

The helpers and the rules are the constructs used to specify the transformation behaviour. The source pattern of the rules (e.g. lines 15-20) consists of a source type and an Object Constraint Language (OCL) [Obj06] guard stating the elements to be matched. Each rule specifies a target pattern (e.g. lines 21-25) which is composed of a set of elements, each of them (as the one at lines 22-25) specifies a target type from the target metamodel (for instance, the type `MatchedRule` from the ATL metamodel) and a set of bindings. A binding refers to a feature of the type, i.e., an attribute, a reference or an association end, and specifies an expression whose value initializes the feature. $\mathcal{H}_R$ consists of a set of rules each of them devoted to the management of one of the resolvable metamodel changes reported in Tab. 6.1. For instance, the Listing 6.1 contains the rules for generating the co-evolution actions corresponding to the *Rename metaelement* and the *Extract metaclass* changes.

Listing 6.1: Fragment of the HOT_R transformation

```
1 module H_R; create OUT : ATL  from Delta : KM3Diff; ... rule
2 atlModule {
3     from
4         s: KM3Diff!Metamodel
5     to
```

```
6       t : ATL!Module (
7            name <- 'CTR',
8            outModels <- Sequence {tm},
9         inModels <- Sequence {sm},...
10      ),...
11  } rule CreateRenaming {
12    from
13       input : KM3Diff!Class,
14       delta : KM3Diff!ChangedClass
15       ...
16       (not input.isAbstract
17        and input.name <> delta.updatedElement.name...)
18    to
19       matchedRule :  ATL!MatchedRule (
20         name<-input.name + '2' + delta.updatedElement.name,
21         ...
22       ),...
23  } rule CreateExtractMetaClass {
24   from
25       cr1: KM3Diff!ChangedReference, cr2: KM3Diff!ChangedReference, r1 : KM3Diff!
             ↪Reference, r2 : KM3Diff!Reference, c1 : KM3Diff!Class,
26       c2 : KM3Diff!Class,...
27       ( cr1.updatedElement = r2 and cr1.owner = c2
28         and cr1.type = c1 and ...)
29   to
30       -- MatchedRule generation
31       matchedRule_i_c2 :  ATL!MatchedRule (
32          name<-i_c2.name + '2' + i_c2.name,
33          inPattern <- ip_i_c2,
34          outPattern <- op_i_c2,
35          ...
36       ),...
37  } ...
```

The application of $\mathcal{H}_R$ to the metamodel $\texttt{MM}_0$ in Fig. 6.1 and the difference model $\Delta_{R(0,1)}$ in Fig. 6.7 generate the model transformation reported in the Listing 6.2. In fact, the source pattern of the `CreateExtractMetaClass` rule (lines 28-32 in the Listing 6.1) matches with the two *Extract metaclass* changes represented in $\Delta_{R(0,1)}$. They consist of the additions of the `PTArc` and `TPArc` metaclasses instead of the direct references between the existing elements `Place` and `Transition`. Consequently, according to the structural features of the involved elements, the `CreateExtractMetaClass` rule generates the transformation $CT_{R(0,1)}$ which is able to co-evolve all the models conforming to $\texttt{MM}_0$ by adapting them with respect to the new metamodel $\texttt{MM}_1$ (see line 1-2 of the Listing 6.2). In particular, each element of type `Place` has to be modified by changing all the references to elements of type `Transition` with references to new elements of type `PTArc` (see lines 4-23 in the Listing 6.2). The same modification has to be performed for all the elements of type `Transition` by creating new elements of type `TPArc` which have to be added instead of direct references between `Transition` and `Place` instances (see lines 24-42).

Listing 6.2: Fragment of the generated $CT_{R(0,1)}$ transformation

```
1   IN : MM0; ... rule Place2Place {
2     from
3       s : MM1!Place
4       ...
5     to
6       t : MM2!Place (
7         name <- s.name,
8         net <- s.net,
9         out <- s.dst->collect(e |
10          thisModule.createPTArc(e, t)
11        )
12      )
13  } rule createPTArc(s : OclAny, n : OclAny) {
14    to
15      t : MM2!PTArc (
```

```
16        src <- s,
17        dst <- n
18      ), ...
19  } rule Transition2Transition {
20    from
21      s : MM1!Transition
22      ...
23    to
24      t : MM2!Transition (
25        net <- s.net,
26        in <- s.dst->collect(e |
27          thisModule.createTPArc(e, t)
28        )
29      )
30  } rule createTPArc(s : OclAny, n : OclAny) {
31    to
32      t : MM2!PTArc (
33        dst <- s,
34        src <- n
35      ), ...
36  } ...
```

The management of the breaking and unresolvable modifications is based on the same techniques presented so far for the breaking resolvable case. However, as mentioned in Sect. 6.1, the involved transformations cannot automatically co-adapt the models but are limited to default actions which have to be refined by the designer.

6.3.2 Parallel dependent changes

As mentioned above, the automatic co-adaptation of models relies on the parallel independence of breaking resolvable and unresolvable modifications, or more formally

$$\Delta_R|\Delta_{\neg R} = \Delta_R; \Delta_{\neg R} + \Delta_{\neg R}; \Delta_R \tag{6.1}$$

where + denotes the non-deterministic choice. In essence, their application is not affected by the adopted order since they do not present any interdependencies. In case the modifications in Tab. 6.1 refer to the same elements, then the order in which such modifications take place matters and does not allow the decomposition of a difference model as, for instance, when evolving $\texttt{MM}_0$ directly to $\texttt{MM}_2$ (although the sub steps $\texttt{MM}_0 - \texttt{MM}_1$ and $\texttt{MM}_1 - \texttt{MM}_2$ are directly manageable as described in the previous section).

A possible approach, which is only sketched in the following, consists in isolating the interdependencies whenever (6.1) does not hold. The intention is to define an iterative process consisting in *diminishing* the modifications between two metamodels until the corresponding breaking resolvable and unresolvable differences are parallel independent. In particular, let Δ be a difference between two metamodels, then we denote by $\mathcal{P}(\Delta)$ the *difference powermodel*, that is the (partially ordered) set of all possible valid sub models of Δ (i.e., fragments of the difference model which are still conforming to the difference metamodel):

$$\mathcal{P}(\Delta) = \{\delta_0 = \phi, \cdots, \delta_i, \delta_{i+1}, \cdots, \delta_n = \Delta\}$$

Then, the solution is the smallest k in $\{0, \cdots, n\}$ such that

$$\Delta^{(k)}; \delta_k = \Delta$$

where $\Delta^{(k)}$ is the difference model between Δ and δ_k, and

$$\Delta^{(k)} = \Delta_R^{(k)}|\Delta_{\neg R}^{(k)}$$

with $\Delta_R^{(k)}$ and $\Delta_{\neg R}^{(k)}$ parallel independent. Hence, the problem of parallel dependence is reduced to the following

$$\Delta = (\Delta_R^{(k)} | \Delta_{\neg R}^{(k)}); \delta_k$$

by applying the higher-order transformation introduced in the previous section. For instance, if we consider $(\mathtt{MM}_2 - \mathtt{MM}_0)$ the solution consists in iteratively finding a difference model which maps $\mathtt{MM}_0$ to the intermediate metamodel corresponding to $\mathtt{MM}_2$ without the attribute *weight* of the `Arc` metaclass. Therefore, the remaining δ_k in this example is a non resolvable change, while in general it may demand further iterations of the decomposition process.

Chapter 7

Conclusion

In this deliverable we presented a model-driven approach to manage the upgrade of FOSS distributions. More specifically, we presented the metamodels on which our approach is based and we showed by means of simple examples how the proposed metamodels allow a reasonable description of the state of the system and representation of its evolution over time.

This approach represents an important result for the Mancoosi project in the following directions:

- it provides the base on which to develop features to complete resolve packages dependencies, also considering package settings. An example of this is introduced in Chapter 2 and is used in Chapter 5 in order to show how the presented approach is able to predict inconsistencies that actual package managers are not able to deal with.
- using this approach, the goal of WP3 is to provide tools and algorithms to keep track of the evolution of the system and to revert the system to previous (working) states and retrieve it in an efficient way. In order to do so, it is extremely important to simulate the execution of maintainer scripts as well. It will be possible by describing maintainer scripts in terms of models.

In fact, the intention is to specify how installation scripts affect the state of a client system in order to treat the upgrades in a transactional way. The metamodels proposed in this deliverable will be the foundation to define a new Domain Specific Language (DSL) to specify the behavior of the scripts. The DSL should be defined according to the modeling primitives given by a coordinated set of metamodels which are formalized in this deliverable. Such metamodels represent the modeling languages for describing system configuration, packages and upgrade history. The definition of the DSL consists of abstract syntax, concrete syntax and dynamic semantics: while the abstract syntax is given through the metamodels, the concrete syntax and the dynamic semantics are among the main objectives of Deliverable D3.2.

Furthermore, we will instantiate the metamodel on a widely used GNU/Linux distribution, according to Deliverable D2.2, and we will implement a model-based framework for managing the complexity and the state of the GNU/Linux instantiation, according to Deliverable D2.3.

Appendix A

Autoscript templates

A.1 Debian `debhelper` autoscript templates

Desktop

Listing A.1: postinst-desktop

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

Listing A.2: postrm-desktop

```
if [ "$1" = "remove" ] && which update-desktop-database >/dev/null 2>&1 ; then
    update-desktop-database -q
fi
```

Doc-base

Listing A.3: postinst-doc-base

```
if [ "$1" = configure ] && which install-docs >/dev/null 2>&1; then
    install-docs -i /usr/share/doc-base/#DOC-ID#
fi
```

Listing A.4: prerm-doc-base

```
if [ "$1" = remove ] || [ "$1" = upgrade ] && \
   which install-docs >/dev/null 2>&1; then
    install-docs -r #DOC-ID#
fi
```

emacs

Listing A.5: postinst-emacsen

```
if [ "$1" = "configure" ] && [ -x /usr/lib/emacsen-common/emacs-package-install ]
then
    /usr/lib/emacsen-common/emacs-package-install #PACKAGE#
fi
```

Listing A.6: prerm-emacsen

```
if [ -x /usr/lib/emacsen-common/emacs-package-remove ] ; then
    /usr/lib/emacsen-common/emacs-package-remove #PACKAGE#
fi
```

gconf

Listing A.7: postinst-gconf-defaults

```
if [ "$1" = "configure" ] && which update-gconf-defaults >/dev/null 2>&1; then
    update-gconf-defaults
fi
```

Listing A.8: postrm-gconf-defaults

```
if which update-gconf-defaults >/dev/null 2>&1; then
    update-gconf-defaults
fi
```

Listing A.9: postinst-gconf

```
if [ "$1" = "configure" ]; then
    gconf-schemas --register #SCHEMAS#
fi
```

Listing A.10: prerm-gconf

```
if [ "$1" = remove ] || [ "$1" = upgrade ]; then
    gconf-schemas --unregister #SCHEMAS#
fi
```

Listing A.11: postrm-gconf

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

icons

Listing A.12: postinst-icons

```
if which update-icon-caches >/dev/null 2>&1 ; then
    update-icon-caches #DIRLIST#
fi
```

Listing A.13: postrm-icons

```
if which update-icon-caches >/dev/null 2>&1 ; then
    update-icon-caches #DIRLIST#
fi
```

info

Listing A.14: postinst-info

```
if [ "$1" = "configure" ]; then
    install-info --quiet #FILE#
fi
```

Listing A.15: prerm-info

```
if [ "$1" = remove ] || [ "$1" = upgrade ]; then
    install-info --quiet --remove #FILE#
fi
```

init

Listing A.16: postinst-init-nostart

```
if [ -x "/etc/init.d/#SCRIPT#" ]; then
    update-rc.d #SCRIPT# #INITPARMS# >/dev/null || #ERROR_HANDLER#
fi
```

Listing A.17: postinst-init-restart

```
if [ -x "/etc/init.d/#SCRIPT#" ]; then
    update-rc.d #SCRIPT# #INITPARMS# >/dev/null
    if [ -n "$2" ]; then
        _dh_action=restart
    else
        _dh_action=start
    fi
    if [ -x "`which␣invoke-rc.d␣2>/dev/null`" ]; then
        invoke-rc.d #SCRIPT# $_dh_action || #ERROR_HANDLER#
    else
        /etc/init.d/#SCRIPT# $_dh_action || #ERROR_HANDLER#
    fi
fi
```

Listing A.18: prerm-init-norestart

```
if [ -x "/etc/init.d/#SCRIPT#" ] && [ "$1" = remove ]; then
    if [ -x "`which␣invoke-rc.d␣2>/dev/null`" ]; then
        invoke-rc.d #SCRIPT# stop || #ERROR_HANDLER#
    else
        /etc/init.d/#SCRIPT# stop || #ERROR_HANDLER#
    fi
fi
```

Listing A.19: postinst-init

```
if [ -x "/etc/init.d/#SCRIPT#" ]; then
    update-rc.d #SCRIPT# #INITPARMS# >/dev/null
    if [ -x "`which␣invoke-rc.d␣2>/dev/null`" ]; then
        invoke-rc.d #SCRIPT# start || #ERROR_HANDLER#
    else
        /etc/init.d/#SCRIPT# start || #ERROR_HANDLER#
    fi
fi
```

Listing A.20: prerm-init

```
if [ -x "/etc/init.d/#SCRIPT#" ]; then
    if [ -x "`which␣invoke-rc.d␣2>/dev/null`" ]; then
        invoke-rc.d #SCRIPT# stop || #ERROR_HANDLER#
    else
        /etc/init.d/#SCRIPT# stop || #ERROR_HANDLER#
    fi
fi
```

Listing A.21: postrm-init

```
if [ "$1" = "purge" ] ; then
    update-rc.d #SCRIPT# remove >/dev/null || #ERROR_HANDLER#
fi
```

Make shared libraries

Listing A.22: postinst-makeshlibs

```
if [ "$1" = "configure" ]; then
    ldconfig
fi
```

Listing A.23: postrm-makeshlibs

```
if [ "$1" = "remove" ]; then
    ldconfig
fi
```

Menu

Listing A.24: postinst-menu-method

```
inst=/etc/menu-methods/#PACKAGE#
if [ -f $inst ]; then
    chmod a+x $inst
    if [ -x "`which update-menus 2>/dev/null`" ]; then
        update-menus
    fi
fi
```

Listing A.25: postrm-menu-method

```
inst=/etc/menu-methods/#PACKAGE#
if [ "$1" = "remove" ] && [ -f "$inst" ]; then chmod a-x $inst ; fi
if [ -x "`which update-menus 2>/dev/null`" ]; then update-menus ; fi
```

Listing A.26: postinst-menu

```
if [ "$1" = "configure" ] && [ -x "`which update-menus 2>/dev/null`" ]; then
    update-menus
fi
```

Listing A.27: postrm-menu

```
if [ -x "`which update-menus 2>/dev/null`" ]; then update-menus ; fi
```

Modules

Listing A.28: postinst-modules

```
if [ "$1" = "configure" ]; then
    if [ -e /boot/System.map-#KVERS# ]; then
        depmod -a -F /boot/System.map-#KVERS# #KVERS# || true
    fi
fi
```

Listing A.29: postrm-modules

```
if [ -e /boot/System.map-#KVERS# ]; then
    depmod -a -F /boot/System.map-#KVERS# #KVERS# || true
fi
```

python

Listing A.30: postinst-python

```
PYTHON=#PYVER#
if which $PYTHON >/dev/null 2>&1 && [ -e /usr/lib/$PYTHON/compileall.py ]; then
    DIRLIST="#DIRLIST#"
    for i in $DIRLIST ; do
        $PYTHON /usr/lib/$PYTHON/compileall.py -q $i
    done
fi
```

Listing A.31: prerm-python

```
dpkg -L #PACKAGE# |
    awk '$0~/\.py$/ {print $0"c\n" $0"o"}' |
    xargs rm -f >&2
```

scrollkeeper

Listing A.32: postinst-scrollkeeper

```
if [ "$1" = "configure" ] && which scrollkeeper-update >/dev/null 2>&1; then
    scrollkeeper-update -q
fi
```

Listing A.33: postrm-scrollkeeper

```
if [ "$1" = "remove" ] && which scrollkeeper-update >/dev/null 2>&1; then
    scrollkeeper-update -q
fi
```

sgmlcatalog

Listing A.34: postinst-sgmlcatalog

```
if [ "$1" = "configure" ]; then
    rm -f #CENTRALCAT#
    for ordcat in #ORDCATS#; do
        update-catalog --quiet --add #CENTRALCAT# ${ordcat}
    done
    update-catalog --quiet --add --super #CENTRALCAT#
fi
```

Listing A.35: prerm-sgmlcatalog

```
if [ "$1" = "remove" ] || [ "$1" = "upgrade" ]; then
    update-catalog --quiet --remove --super #CENTRALCAT#
fi
```

Listing A.36: postrm-sgmlcatalog

```
if [ "$1" = "purge" ]; then
    rm -f #CENTRALCAT# #CENTRALCAT#.old
fi
```

mime

Listing A.37: postinst-sharedmimeinfo

```
if [ "$1" = "configure" ] && [ -x "`which␣update-mime-database␣2>/dev/null`" ]; then
    update-mime-database /usr/share/mime
fi
```

Listing A.38: postrm-sharedmimeinfo

```
if [ -x "`which␣update-mime-database␣2>/dev/null`" ]; then
    update-mime-database /usr/share/mime
fi
```

Listing A.39: postinst-mime

```
if [ "$1" = "configure" ] && [ -x "`which␣update-mime␣2>/dev/null`" ]; then
    update-mime
fi
```

Listing A.40: postrm-mime

```
if which update-mime >/dev/null 2>&1; then update-mime; fi
```

suid

Listing A.41: postinst-suid

```
if [ "$1" = "configure" ]; then
    if which suidregister >/dev/null 2>&1 && [ -e /etc/suid.conf ]; then
            suidregister -s #PACKAGE# /#FILE# #OWNER# #GROUP# #PERMS#
    elif [ -e /#FILE# ]; then
            chown #OWNER#:#GROUP# /#FILE#
            chmod #PERMS# /#FILE#
    fi
fi
```

Listing A.42: postrm-suid

```
if [ "$1" = remove ] && [ -e /etc/suid.conf ] && \
   which suidunregister >/dev/null 2>&1; then
        suidunregister -s #PACKAGE# /#FILE#
fi
```

udev

Listing A.43: preinst-udev

```
if [ "$1" = install ] || [ "$1" = upgrade ]; then
    if [ -e "#OLD#" ]; then
        if [ "`md5sum␣\"#OLD#\" | sed -e \"s/ .*//\"`" = \
             "`dpkg-query␣-W␣-f='${Conffiles}'␣#PACKAGE#␣|␣sed␣-n␣-e␣\"\\\\' #OLD#'s/.*
                 ↪//p\"`" ]
        then
            rm -f "#OLD#"
        fi
    fi
    if [ -L "#RULE#" ]; then
        rm -f "#RULE#"
    fi
fi
```

Listing A.44: postinst-udev

```
if [ "$1" = configure ]; then
    if [ -e "#OLD#" ]; then
        echo "Preserving␣user␣changes␣to␣#RULE#␣..."
        if [ -e "#RULE#" ]; then
            mv -f "#RULE#" "#RULE#.dpkg-new"
        fi
        mv -f "#OLD#" "#RULE#"
    fi
fi
```

usrlocal

Listing A.45: postinst-usrlocal

```
if [ "$1" = configure ]; then
(
    while read line; do
        set -- $line
        dir="$1"; mode="$2"; user="$3"; group="$4"
        if [ ! -e "$dir" ]; then
            if mkdir "$dir" 2>/dev/null; then
                chown "$user":"$group" "$dir"
                chmod "$mode" "$dir"
            fi
        fi
    done
) << DATA
#DIRS#
DATA
fi
```

Listing A.46: prerm-usrlocal

```
(
    while read dir; do
        rmdir "$dir" 2>/dev/null || true
    done
) << DATA
#JUSTDIRS#
DATA
```

wm

Listing A.47: postinst-wm-noman

```
if [ "$1" = "configure" ]; then
    update-alternatives --install /usr/bin/x-window-manager \
        x-window-manager #WM# #PRIORITY#
fi
```

Listing A.48: postinst-wm

```
if [ "$1" = "configure" ]; then
    update-alternatives --install /usr/bin/x-window-manager \
        x-window-manager #WM# #PRIORITY# \
        --slave /usr/share/man/man1/x-window-manager.1.gz \
        x-window-manager.1.gz #WMMAN#
fi
```

Listing A.49: prerm-wm

```
if [ "$1" = "remove" ]; then
    update-alternatives --remove x-window-manager #WM#
fi
```

xfonts

Listing A.50: postinst-xfonts

```
if which update-fonts-dir >/dev/null 2>&1; then
    #CMDS#
fi
```

Listing A.51: postrm-xfonts

```
if [ -x "`which update-fonts-dir 2>/dev/null`" ]; then
#CMDS#
fi
```

debconf

Listing A.52: postrm-debconf

```
if [ "$1" = purge ] && [ -e /usr/share/debconf/confmodule ]; then
    . /usr/share/debconf/confmodule
    db_purge
fi
```

A.2 Fedora "autoscript" snippets

gconf

Listing A.53: pre-gconf

```
%pre
if [ "$1" -gt 1 ] ; then export
GCONF_CONFIG_SOURCE=gconftool-2 --get-default-source
gconftool-2 --makefile-uninstall-rule \
%{_sysconfdir}/gconf/schemas/[NAME] .schemas >/dev/null || :
fi
```

Listing A.54: post-gconf

```
%post
export GCONF_CONFIG_SOURCE=gconftool-2
--get-default-source gconftool-2 --makefile-install-rule \
%{_sysconfdir}/gconf/schemas/[NAME] .schemas > /dev/null ||| :
```

Listing A.55: preun-gconf

```
%preun
if [ "$1" -eq 0 ] ; then export
GCONF_CONFIG_SOURCE=gconftool-2 --get-default-source
gconftool-2 --makefile-uninstall-rule \
%{_sysconfdir}/gconf/schemas/[NAME] .schemas > /dev/null || :
fi
```

Listing A.56: postun-gconf

```
%postun
export GCONF_CONFIG_SOURCE="$(gconftool-2␣--get-default-source)"
gconftool-2 --makefile-uninstall-rule %{_sysconfdir}/gconf/schemas/%{name}.schemas &>/
    ↪dev/null || :
```

info

Listing A.57: post-texinfo

```
%post
/sbin/install-info %{_infodir}/%{name}.info %{_infodir}/dir || :
```

Listing A.58: preun-texinfo

```
%preun
if [ $1 = 0 ] ; then
  /sbin/install-info --delete %{_infodir}/%{name}.info %{_infodir}/dir || :
fi
```

scrollkeeper

Listing A.59: post-scrollkeeper

```
%post
scrollkeeper-update -q -o %{_datadir}/omf/%{name} || :
```

Listing A.60: postun-scrollkeeper

```
%postun
scrollkeeper-update -q || :
```

Desktop

Listing A.61: post-desktop-database

```
%post
update-desktop-database &> /dev/null || :
```

Listing A.62: postun-desktop-database

```
%postun
update-desktop-database &> /dev/null || :
```

mime

Listing A.63: post-mimeinfo

```
%post
update-mime-database %{_datadir}/mime &> /dev/null || :
```

Listing A.64: postun-mimeinfo

```
%postun
update-mime-database %{_datadir}/mime &> /dev/null || :
```

icons

Listing A.65: post-icon

```
%post
touch --no-create %{_datadir}/icons/hicolor
if [ -x %{_bindir}/gtk-update-icon-cache ] ; then
  %{_bindir}/gtk-update-icon-cache --quiet %{_datadir}/icons/hicolor || :
fi
```

Listing A.66: postun-icon

```
%postun
touch --no-create %{_datadir}/icons/hicolor
if [ -x %{_bindir}/gtk-update-icon-cache ] ; then
  %{_bindir}/gtk-update-icon-cache --quiet %{_datadir}/icons/hicolor || :
fi
```

xfonts

Listing A.67: post-fonts

```
%post
if [ -x %{_bindir}/fc-cache ] ; then
  %{_bindir}/fc-cache %{_datadir}/fonts || :
fi
```

Listing A.68: postun-fonts

```
%postun
if [ "$1" = "0" ] ; then
  if [ -x %{_bindir}/fc-cache ] ; then
    %{_bindir}/fc-cache %{_datadir}/fonts || :
  fi
fi
```

usrlocal

Listing A.69: pre-user

```
%pre
getent group GROUPNAME >/dev/null || groupadd -r GROUPNAME
getent passwd USERNAME >/dev/null || \
useradd -r -g GROUPNAME -d HOMEDIR -s /sbin/nologin \
-c "Useful␣comment␣about␣the␣purpose␣of␣this␣account" USERNAME
exit 0
```

Make shared libraries

Listing A.70: post-sharedlibs

```
%post
/sbin/ldconfig
```

Listing A.71: postun-sharedlibs

```
%postun
/sbin/ldconfig
```

It is also common to invoke these shared libraries scripts with the '-p' option as they are often the only program invoked in a scriptlet:

Listing A.72: post-sharedlibs-p

```
%post -p /sbin/ldconfig
```

Listing A.73: postun-sharedlibs-p

```
%postun -p /sbin/ldconfig
```

init

Listing A.74: post-initscript

```
%post
# This adds the proper /etc/rc*.d links for the script
/sbin/chkconfig --add <script>
```

Listing A.75: preun-initscript

```
%preun
if [ $1 = 0 ] ; then
    /sbin/service <script> stop >/dev/null 2>&1
    /sbin/chkconfig --del <script>
fi
```

Listing A.76: postun-initscript

```
%postun
if [ "$1" -ge "1" ] ; then
    /sbin/service <script> condrestart >/dev/null 2>&1 || :
fi
```

A.3 Mandriva macros

info

Listing A.77: Info post macro

```
\%post
\%_install_info \%\{name\}.info
```

Listing A.78: Info preun macro

```
\%preun
\%_remove_install_info
\%\{name\}.info
```

Menu

Listing A.79: update post menu macro

```
1 \%post
2 \%{update_menus}
```

Listing A.80: update postun menu macro

```
1 \%postun
2 \%{clean_menus}
```

init

Listing A.81: initscript post macro

```
1 \%post
2 \%_post_service \<initscript-name\>
```

Listing A.82: initscript preun macro

```
1 \%preun
2 \%_preun_service \<initscript-name\>
```

ghostfile

Listing A.83: ghostfile post macro

```
1 \%post
2 \%create_ghostfile /\var/lib/games/powermanga.hi root games 664
```

Where the %create_ghostfile macro will expand to:

Listing A.84: create_ghostfile

```
1 \if \[ \! -f /\var/lib/games/powermanga.hi \]; then
2   touch /\var/lib/games/powermanga.hi
3   chown root.games /\var/lib/games/powermanga.hi
4   chmod 664 /\var/lib/games/powermanga.hi
5 \fi
```

Desktop

Listing A.85: update-desktop-database post macro

```
1 %post
2 %update_desktop_database
```

Listing A.86: update-desktop-database postun macro

```
1 %postun
2 %clean_desktop_database
```

mime

Listing A.87: update-mime-database post macro

```
1 %post
2 %update_mime_database
```

Listing A.88: update-mime-database postun macro

```
1 %postun
2 %clean_mime_database
```

icons

Listing A.89: icon post macro

```
%post
%update_icon_cache hicolor
%update_icon_cache crystalsvg
```

Listing A.90: icon postun macro

```
%postun
%update_icon_cache hicolor
%update_icon_cache crystalsvg
```

gconf

Listing A.91: GConf post macro

```
%post
%post_install_gconf_schemas %{schemas}
```

Listing A.92: GConf preun macro

```
%preun
%preun_uninstall_gconf_schemas %{schemas}
```

scrollkeeper

Listing A.93: scrollkeeper post macro

```
%post
%update_scrollkeeper
```

Listing A.94: scrollkeeper postun macro

```
%postun
%clean_scrollkeeper
```

Bibliography

[AKB02] M. Aksit, I. Kurtev, and J. Bézivin. Technological Spaces: an Initial Appraisal. International. Federated Conf. (DOA, ODBASE, CoopIS), Industrial Track, Los Angeles, 2002.

[ATL] ATLAS Group. The Atlantic Zoo. `http://www.eclipse.org/gmt/am3/~zoos/atlanticZoo/`.

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

[Bai97] Bailey, Edward C. *Maximum RPM: Taking the Red Hat Package Manager to the Limit*. Sams; 1st edition (August 16, 1997): http://onlinebooks.library.upenn.edu/webbin/book/lookupid?key=olbp11202, 1997.

[BAS09] Bash shell. `http://www.gnu.org/software/bash/`, Last visited January 2009.

[BG01] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Automated Software Engineering (ASE 2001)*, pages 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BJRV04] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture, European MDA Workshops: Foundations and Applications*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.

[Bou] Stephen R. Bourne. An introduction to the unix shell. Unix Seventh Edition Manual, Volume 2. (1979).

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

[BTLd] Paulo Barata, Paulo Trezentos, Inês Lynce, and Davide di Ruscio. Mancoosi deliverable d3.1: Survey of the state of the art technologies for handling versioning, rollback and state snapshot in complex systems. http://www.mancoosi.org/deliverables/d3.1.pdf.

[CDEP08] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *12th IEEE International EDOC Conference (EDOC 2008)*, pages 222–231, München (Germany), 2008. IEEE Computer Society.

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

[Cic08] A. Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, University of L'Aquila, Computer Science Dept., 2008.

[Cou06] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2), 2006.

[CRP08] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing model conflicts in distributed development. In *MoDELS 2008*, volume 5301 of *LNCS*, pages 311–325, 2008.

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

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

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

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

[Fav03] Jean-Marie Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *Procs. of the Int. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM 2003*, Amsterdam, September 2003.

[GKP07] B. Gruschko, D. Kolovos, and R. Paige. Towards Synchronizing Models with Evolving Metamodels. In *Proceedings of the Workshop on Model-Driven Software Evolution (MODSE 2007)*, 2007.

[JB06] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *FMOODS'06*, volume 4037 of *LNCS*, pages 171–185. Springer-Verlag, 2006.

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

[KW03] A. Kleppe and J. Warmer. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

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

[LZG04] Yuehua Lin, Jing Zhang, and Jeff Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, 2004.

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

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

[MCF03] S. J. Mellor, A. N. Clark, and T. Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, 2003.

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

[MSD06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *MoDELS 2006*, volume 4199 of *LNCS*, pages 200–214, 2006.

[MZ07] Karl Mazurak and Steve Zdancewic. Abash: finding bugs in bash scripts. In *PLAS '07*, pages 105–114. ACM, 2007.

[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.

[Obj02] Object Management Group (OMG). MOF 2.0 Query/Views/Transformations RFP, 2002. OMG document ad/02-04-10.

[Obj03a] Object Management Group (OMG). MDA Guide version 1.0.1, 2003. OMG Document: omg/2003-06-01.

[Obj03b] 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.

[Obj03c] Object Management Group (OMG). UML 2.0 Infrastructure Final Adopted Specification, 2003. OMG document ptc/03-09-15.

[Obj03d] Object Management Group (OMG). XMI 2.0 XML Metadata Interchange, 2003. OMG document formal/2003-05-02.

[Obj06] Object Management Group (OMG). OCL 2.0 Specification, 2006. OMG Document formal/2006-05-01.

[Oli04] James Olin Oden. Transactions and rollback with rpm. *Linux Journal*, 2004(121):1, 2004.

[PER09] The perl directory. `http://www.perl.org/`, Last visited January 2009.

[RV08] J.E. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *TOOLS EUROPE 2008, 46th Intl. Conf. Objects, Models, Components, Patterns*, Zurich, Switzerland, 2008. To appear.

[Sch06a] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Sch06b] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, September/October 2003.

[Sel03] B. Selic. The Pragmatics of Model-driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SS04] Diomidis Spinellis and Clemens Szyperski. How is open source affecting software development. *IEEE Computer*, 21(1):28–33, 2004.

[Szy98] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Szy03] Clemens Szyperski. Component technology: what, where, and how? In *Proceedings of ICSE03.* ACM, 2003.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 295–304, New York, NY, USA, 2007. ACM.

[TDL+07] Paulo Trezentos, Roberto Di Cosmo, Stephane Lauriere, Mario Morgado, Joao Abecasis, Fabio Mancinelli, and Arlindo Oliveira. New Generation of Linux Meta-installers. *Research Track of FOSDEM 2007*, 2007.

[Tou] A. Toulmé. The EMF Compare Utility. `http://www.eclipse.org/modeling/emft/`.

[TSJL07] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *ICSE '07*, pages 178–188. IEEE Computer Society, 2007.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4069 of *Lecture Notes in Computer Science.* Springer-Verlag, July 2007.

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