First version of the DSL based on the model developed in WP2

Deliverable 3.2

Nature : Deliverable
Due date : 1.11.2009
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> John Thomson <john.thomson@caixamagica.pt> Patrizio Pelliccione <pellicci@di.univaq.it> Alfonso Pierantonio <alfonso@di.univaq.it>
Internal review	Jeff Johnson <n3npq@mac.com> David Lutterkort <lutter@watzmann.net>
Workpackage number	WP3
Deliverable number	2
Document type	Deliverable
Version	1
Due date	01/11/2009
Actual submission date	01/11/2009
Distribution	Public
Project coordinator	Roberto Di Cosmo <roberto@dicosmo.org>

Preface

This document has been reviewed by two experts from industry working in the package management area. They were chosen specifically for their experience and knowledge of topics that are addressed in this document. By doing this, we can be relatively assured that the proposed mechanisms and the DSL have covered sufficient grounds for the first version. Another reason for this selection is that they have detailed knowledge of implementing DSLs in practical systems and as such allow us to be confident that the approach can be adopted by industry and not be solely a research topic.

We would like to acknowledge their help and feedback which was invaluable not only for input into this document but also for enhancing our approach and how we will implement it in the scope of the rest of the Mancoosi project.

Abstract

Today's software systems require an evolution in order to both satisfy stakeholders' increasing requirements and to react to possible faults. The advancement cannot be confined to phases, as it happens continuously. Software systems are typically not considered monolithic but organized in *packages*, brought to popularity by Free and Open Source Software (FOSS) *distributions*. Packages promote the philosophy of system evolution, which typically is implemented as *upgrades*, but they do not solve all management problems of large software collections. These problems are mainly caused by implicit inter-package dependencies that cannot be handled by upgrades that are typically non-transactional.

This deliverable presents a first version of the Domain Specific Language (DSL) based on the *model-driven* approach presented in WP2 to support the upgrades of FOSS distributions. Both static and dynamic aspects of packages are taken into account: the former relies on implicit inter-package dependencies, whereas the latter depends on the execution of specific scripts which are executed during package upgrades. Scripts are implemented in Turing-complete languages, and all non-trivial properties about them are undecidable, including determining a priori their effects to be able to revert them upon failure. In this respect, a new model-based Domain Specific Language is provided for specifying and to simulate the behavior of the scripts which are executed during package upgrades. There are two benefits of this approach: *i)* by simulating the upgrade on the models, our proposed technique is able to *predict* some upgrade failures, and *ii)* the models and transactional logs can drive at run-time the roll-back of residual effects of failed or undesired upgrades. The aim of creating the DSL is to be able to provide the language to capture the atomic operations performed and modelled in such a way that we can simulate the effect of transactional upgrades and roll-backs before applying them to the system. The effectiveness of the approach is investigated by applying it to some Linux distributions.

Conformance

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

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

Contents

List of Figures

List of Acronyms

ACID Atomicity Consistency Isolation Durability

ATL ATLAS Transformation Language

CUDF Common Upgradeability Description Format

DSL Domain Specific Language

DUDF Distribution Upgradeability Description Format

FOSS Free and Open Source Software

GPL General Purpose Language

MANCOOSI Managing Software Complexity

MDE Model Driven Engineering

MOF Meta Object Facility

OMF Open Source Metadata Framework

OMG Object Management Group

POSIX Portable Operating System Interface [for Unix]

UML Unified Modeling Language

XML XML Metadata Interchange

Chapter 1

Introduction

Modern software systems are characterized by an ever increasing need to integrate new capabilities, solutions to requirements and to drop unused ones. This evolution is in response to "changing user needs, system intrusion or faults, changing operational environments, and resource variability" [CGI$^+$08]. The growth cannot be confined in to phases, as it happens continuously and incrementally. Therefore, mechanisms for run-time evolution are needed to provide good reactions to both system and contextual changes. The evolution should be guided and constrained by rules that allow local needs to be satisfied in local ways and by preserving the integrity of the overall system [ea06].

1.1 Background

FOSS systems are among the most complex software systems known, made of many components that mature over time without centralized design: components advance independently from each other. They provide, today, a real-world example of what tomorrow's complex, quickly changing software systems will look like. In fact FOSS systems are typically written by small, independent teams that, without central coordination, are dedicated to developing independent software applications [DH07]. More precisely, FOSS systems are typically organized in *packages*, i.e., in a format that can be easily processed by automatic tools and that contains some additional information useful to handle their installation, removal and update. In a limited allegory this approach is similar to that of a car manufacturing facility where each sub-component is designed, created and tested as individual sub-components of a larger system. Rather than re-inventing the wheel each time, the logical blocks are separated and can be worked on separately and optimised as individual units. Each package normally provides a small piece of a puzzle as they are logical, functional blocks that are then integrated by larger systems that can make use of the pipelined structure. This promotes better code-reusability and the optimisation of smaller units and it also means that the fragmented development system is possible and allows for distributed code projects like GNU/Linux to exist.

Maintenance of FOSS systems involves not only the classical activities of correction, adaptation, and perfection [Swa76], but also integration of new versions of software that have been released. Packages promote the system evolution, which is typically implemented as *upgrades*, but they do not solve all management problems of large software collections. These problems are mainly caused by implicit inter-package dependencies that cannot be handled by upgrades that are typically non-transactional. In fact current approaches are able to deal only with static aspects but not with dynamic ones that can cause implicit inter-package dependencies. It is not rare therefore to make the system unusable or unstable by installing or removing some packages that compromise the stability of the system.

The Mancoosi[1] project aims at defining a *model-driven* approach to support the upgrades and transactional rollbacks of FOSS distributions. The approach should be able to model and manage both static and dynamic aspects of the approach. The dynamic aspect of the packages is realized and described by means of a set of executable maintainer (also known as "configuration") scripts that are executed during the upgrade and perform operations required pre or post installation or removal of each single package. The most common use case for maintainer scripts is to update some cache, blending together data shipped by the package, with data installed on the system, possibly from other packages. Maintainer scripts are written in Turing-complete programming languages and they often rely on data which is available only on the target installation machine, and not in the package itself. It is clear then that writing maintainer scripts is a task susceptible to errors that requires diligence on the part of the package maintainers. Moreover, typically upgrade failures are usually detected via inspection of script exit code without exact information of the configuration reached and of operations already performed. A number of packages will only display warnings so not to interrupt the installation. Therefore, once a possible erroneous configuration has been reached the user might not be aware about the possible erroneous behaviors and is not warned of the need or guided to reach a new correct configuration (hopefully reverting back to the system's exact configuration before starting the upgrade).

The adopted scripting languages used to write maintainer scripts have rarely been formally investigated, thus posing additional difficulties in understanding their side-effects which can spread throughout the system. State of the art package managers lack several important features such as complete dependency resolution and roll-back of failed upgrades. Moreover, there is no support to simulate upgrades taking the behavior of maintainer scripts into account. In fact, current tools consider only inter-package relationships which are not enough to predict side-effects and system inconsistencies which can be encountered during upgrades. In order to deal with these challenging aspects of the upgrade of FOSS systems, in this document we propose a model-based DSL which will be *distribution independent* for writing maintainers scripts. The DSL forces distribution maintainers to use a subset of high level statements (a sort of macro) with well defined semantics. The well defined semantics enable a deep understanding of how the system evolved in to a new configuration through the execution of a single statement. Furthermore the DSL is model-based, i.e., its abstract syntax is defined in terms of the Mancoosi metamodel (see the deliverable D2.1 [2]), and this means that it is possible to hypothesise what the effects will be of an upgrade directly on the models that represent a system execution. This opens new potential scenarios:

- *simulation*: an upgrade can be simulated on the models which represent the system configuration before performing the real upgrade. Since the models contain also the description of the scripts, in terms of DSL statements, the simulation is not only a resolution of explicit dependencies among packages, but it deals also with implicit dependencies and with failures that can occur because of fallacious or missing script statements. There are two possible outputs of the simulation: *not valid* and *valid*. If the simulation is not valid then the real upgrade will fail. If the result of the simulation is valid then, due to the abstraction of the models, the real upgrade can still fail. Also in this case the DSL and

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

[2] `http://www.mancoosi.org/deliverables/d2.1.pdf`

the model-driven approach in general will play its role, as described in the following item;

- *models guide the down-grade*: models store each executed statement thanks to a log model and are continuously synchronized with the system execution. In other words the system's configuration is kept synchronised with the modelled system. The synchronisation will occur when the model is about to be accessed by updating the meta-classes and information that the model will access. Synchronisation of the model is intrinsic to the behaviour of the model and as such will be defined in WP2. Due to the well defined semantics of the corresponding ATL transformations of the configuration scripts and to the detailed knowledge of the effect that a statement execution has on the system's configuration, models can guide the roll-back. Having the current state of the system's configuration and the transaction log of the changes made, the model can then use this information and if there is a reverse operation for each element contained within the transaction log then the model can perform a roll-back for a single package upgrade path. The combination of the log and the model will also be able to store historical information regarding the life-cycle of a package and can drive the roll-back to a set of criteria defined by the user.

The DSL we are proposing derives from a deep analysis of several popular GNU/Linux distributions. This analysis was aimed at finding in maintainer scripts the parts which were automatically generated and thus discovering common generated code recurrences. The language is intended to be extensible since when new commonalities or recurrent failures are identified, new constructs in the language have to be added in order to enable simulations and failure detection for the newly found recurrences. In other words, the DSL and the simulators, can be extended/refined once new "common failures" have been identified.

This document does not provide details about simulators; that will be part of the deliverable D2.3 which is due by the end of the project. This deliverable provides the reader with a general picture about how and what we intend to simulate and present in depth all aspects of the DSL. In particular, both the syntax and semantics of the language are provided. Concerning the latter, we define the operational semantics of the language to describe what happens in a system when a program of that language is executed. In this respect, the runtime system is defined in terms of a configuration model which represents the current state of the system in which a given maintainer script has to be executed. Both the configuration and the script models conform to the Mancoosi meta-model. The semantic mappings which specify the semantics are given in terms of ATLAS Transformation Language (ATL) transformations. In particular, for each command of the DSL a corresponding transformation rule is defined in order to describe how the considered command changes the current configuration model. Some explanatory examples of maintainer script specifications given with the DSL are also provided.

1.2 Related Works

There are currently quite a few types of systems that perform transactional roll-backs and allow a previous configuration of a system to be restored. Most rely on the methods of taking snapshots of the system (some are more efficient in terms of how they realise this) and then when a user requests or a failure condition is met, a previous configuration set is restored. There are many issues that a DSL would be able to resolve that the traditional "copy and paste" type of systems would not be able to offer. One of these is that if any files are located in the same directory as any of the monitored configuration files that there is a chance that it will be removed by the snapshot mechanism depending on the granularity that is used by the system for restoring previous file-system states. Some systems will monitor individual files for changes (e.g. precise granularity) whereas others will monitor a folder that can contain many different types of files (lower granularity). Another issue is that when a previous configuration set is restored it normally reverts back to a previous time and any changes between that time and the current time, even if they affect other configuration files, are lost. For instance, a package A may have been upgraded from version 1 to 2 and at a later stage a package B upgraded from 3 to version 5. Reverting to a previous state to restore package A to version 1 would in many cases entail package B also reverting back to version 3. This often may not be the desired solution, especially when looking to downgrade a single package but may be useful in the case of a catastrophic failure where multiple configurations have been changed and reverting a collection of packages is necessary (in the case of viruses or malicious activity for instance). Being able to revert a single package also means that the current process of maintaining a specific older version of a file as another package and then the user removing a package completely and re-installing the older package would become redundant.

Journalled File Systems are systems where the changes to the hard-drive are first written to a specifically chosen location on the hard-drive. Journal File Systems, to avoid performance issues related to writing twice for each operation,tend to only journal meta-data and not all the file-system data but this is sometimes configurable. They also record all the changes that the file system intends to make. If a crash then occurs at any stage, the correct state of the filesystem can be recovered by replaying the journal data over the original data until it succeeds or by reverting to the last completed transaction. This mechanism relies on the atomicity of transactions and for the rest of the system to deem whether a process has succeeded or not. This however is not the case with most Journal File Systems as the journal data is not validated in many implementations and only the commit block has to have been written correctly for a journal to be re-processed after a crash. A JFS however does not guarantee that the changes stored in the journal are atomic. If all the processes in a journal succeed and the journal data is not corrupt then it can be assumed that the hard-drive has performed the change correctly and is in the correct state. The journal is normally implemented as a buffer that can dynamically change in size. There are many similarities between this methodology and that which the DSL and the simulation use, as explained in the remainder of the document.

ACID is an acronym and set of design principles usually used in conjunction with databases and is used to guarantee that transactions occur reliably. It represents the thoughts and ideas behind over 30 years of work into reliable database design. The complexity of enabling such properties has meant that only recently are compliant systems being produced. Even then there are different levels of compliance. What is important though is that the design principles ensure a deterministic and stateful representation of data. By using these principles for storing the transactional logs and any other sub-systems generated as part of the simulator design in Deliverable 2.3 and Work Package 2 we can ensure that our logs are reliable. The first initial, A, stands for atomicity and is used to ensure that either a complete transaction occurs or not at all. Any combination of results that does not lead to either of these two states are deemed invalid and the system will try to ensure that either the transaction is successful or rollback to the state prior to the transaction. In the case of the simulator the two states map directly into “valid” and “not valid”. C stands for consistency in this case and ensures that only valid data are used in the database. As the simulator will have boundaries it may be possible to ensure consistency but it is more suited in the database domain. I means isolation for this acronym and makes sure that other processes cannot use the data while it is being locked down. APT and most current meta-installers achieve this by locking down the lists files such that only one installer can run at a time. The method of locking and unlocking files is also sometimes used for versioning systems. D is for durability and in databases it means that a transaction is only deemed successful when it is in the transactional log. At this point the changes have been locked down and the state is thought to be stable and will persist.

Each of the aforementioned areas have a similarity in the model that we are proposing and are all used to ensure that only valid states are reached and maintained. By being in a known state and by logging the transactions, each process if it is expressive enough and has one to one functionality will be reversible. A transaction log that is expressive enough will allow for a one to many transaction to be reversed by allowing for the system to know which path to take when reversing the procedure.

In the rest of the section some of the most representative examples of existing systems that perform transactional upgrades are summarised.

ZFS/Nexenta, Nexenta is an operating system first released in 2005. It combines an OpenSolaris Kernel with a GNU application userland. The distribution is based on Ubuntu and aims to provide the ease of use and large repository available through APT of a GNU/Linux based system with some powerful features that Solaris has at its Kernel. ZFS, which used to be known as Zettabyte File System, is a filesystem technology that has been designed to remove a lot of the fundamental limitations in current file-systems and to take advantage of the latest research and technologies. It is licensed under the CDDL but ZFS as a name is a trademark owned by Sun. ZFS exists on top of zpools instead of on a physical drive and so do not need a volume manager to use additional devices. The fundamental limits of physics were used when calculating the upper limits of ZFS so that no other system would ever have to replace it. ZFS because its license is incompatible with GPLv2 cannot be included in most GNU/Linux distributions, but can be run through FUSE, (File System in User Space). This has slowed the uptake of the system but it still contains some very interesting features when looking at file transactions and rollback mechanisms. The main technologies that are of interest in the scope of this project include the copy-on-write system. This is not the only file-system that uses this feature and it is an interesting feature. As hard disk capacities have continued to grow and the cost of storage has decreased with time it has become possible to use hard-disk space to store different configurations and snapshots of system states. Microsoft uses System Restore, Mac OS uses Time Machine and even in GNU/Linux previous kernel installations are maintained. The difference with copy-on-write is that instead of having a lot of replicated information that is common to two or more systems, a clone is only created when a copy procedure is issued. Many callers can therefore have simultaneous access to the same resource and only when a caller modifies it, is the data copied in a way that the other resources can still access the original data.

LVM Snapshots, Logical Volume Management is a form of storage virtualisation. For each logical extent in an LVM system, copy-on-write is performed. Most logical extents are normally mapped to a single physical entity and multiple physical entities can exist on a physical volume, eg. a hard drive. A private copy is only created when the caller tries to modify the data. The volume manager will copy the Logical Extent to a copy-on-write table just before it is written to. This preserves the old Logical Volume as a snapshot which can then be reconstructed by overlaying the copy-on-write table on to the Logical Volume to which it is associated. Read-write snapshots are branching snapshots because the implicitly allow diverging versions of a logical volume. This system is used in ZFS for integrity checking and automatic repair.

NixOS, this operating system is a purely functional variant of Linux. Only deterministic and repeatable Nix expressions are allowed and the realisation of a configuration is not stateful. NixOS has less of the traditional directory structure common to GNU systems and contains symlinks to certain directories and executables such as /bin/sh. Most objects in /etc are symlinks in

NixOS to the store, /nix/store. The only exceptions to this are files that contain mutable states. Sometimes like for /etc/passwd no equivalent exists. The method of using purely functional executables is not a new idea. Other systems are looking at similar technologies. One example of a stateless system is that being investigated by Fedora [3]. *etckeeper,* This collection of tools allows /etc to be stored in git, mercurial, darcs or a bzr repository[4]. The tools hook into apt, yum, pacman-g2 to automatically commit changes made to /etc during package upgrades. Also in addition to standard revision control, the meta-data is also stored and kept in the revision control system. *cowbuilder,* is a possible helper utility and replacement to the debhelp tool. cowbuilder works with multiple distributions and architectures for Debian. It uses chroots/pbuilder/sbuild users to allow pbuilder to build packages without the compress/uncompress stage by using copy-on-write mechanisms.

Conary, created by rPath and distributed under the Common Public License, focuses on installing packages through automated dependency resolution against distributed online repositories. It provides a concise and easy to use Python based description language to specify how to build a package. Foresight and rPath Linux both use conary. Using it, only the updated files in packages are downloaded and this minimises the bandwidth and time requirements for updating the software. Conary also has the feature of allowing rollbacks of a package and as well as derived packages. It introduces the notion of dynamic tags to replace the maintainer script files [5]. These tags are then analysed by conary to see if there are any similar operations to be performed by the packages to be installed and groups these instructions together. Creating tags is a way of grouping similar package installation procedures but just pushes the same issue one level higher. Having a DSL that analyses the maintainer script files can perform this categorisation so this step would become redundant. What is interesting to note though is that the developers are attempting to manage installation transactions rather than individual packages. Other systems perform similar mechanisms and we address this issue in Chapters 2 and 5.

Puppet [6], is a system for managing and administering configurations of large set of systems. Moreover, it provides a means to describe IT infrastructure as policy, execute that policy to build services then audit and enforce ongoing changes to the policy. There is the possibility of adopting the DSL that will be proposed in the rest of the document for achieving similar policy frameworks.

Augeas[7], it is an alternative domain specific language that consider the problem of distributed configuration files within a system and organizing them into a tree based structure. Changes to the configuration files are captured in terms of tree manipulations. These manipulations can then be stored back into the original system. We envisage that in the first version of our DSL we will not need to store configuration changes for maintainer scripts.

1.3 Structure of the deliverable

This deliverable is structured in six chapters:

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

[3] http://fedoraproject.org/wiki/StatelessLinux
[4] http://joey.kitenet/code/etckeeper
[5] http://www.rpath.com/technology/techoverview/otherconcepts.html#dynamictags
[6] http://puppet.reductivelabs.com
[7] http://augeas.net/

- Chapter 2 goes into more detail about the types of failures that can occur during the installation of a package and highlights some of the cases which cannot be dealt with by current meta-installers;
- Chapter 3 explains the methodology and approach chosen. The various parts of the system model are described with reference to Deliverable 2.3 that will provide the concrete definition. How we use the model is expressed within this chapter;
- Chapter 4 plays a key role in this document since it describes the proposed DSL in depth and defines its dynamic semantics by means of ATL transformations;
- Chapter 5 works through studies identified in Chapter 2 and uses the first definition of the language outlined in Chapter 4 to see how the failures could be identified by the model proposed;
- Chapter 6 concludes the Deliverable and describes the main findings through the analysis of the problem domain and the aspects discovered in the creation of the DSL.

1.4 Glossary

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

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

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

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

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

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

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

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

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

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

Chapter 2

Classification of upgrade failures

FOSS distributions, as well as other complex systems, provide their software components in “packaged” form. Packages, available from remote repositories, are installed and removed on local machines by means of package manager applications, such as APT, RPM, URPMI, IPKG or YUM. Package managers are responsible for both finding suitable upgrade strategies by solving dependencies and conflicts among packages, and for actually deploying the resolved set of packages on to the filesystem, possibly aborting the operation if problems are encountered.

During the installation and removal of a package, actions are required in addition to simple file relocation to finalise the component within the overall system configuration. Such actions are usually delegated to executable maintainer scripts, contained in the packages. Maintainer scripts are written mostly in POSIX shell script that makes it very hard, generally impossible, to predict a-priori their side-effects which can affect the entire system. As a consequence, a satisfactory solution able to deal with automatic recovery of faults caused by mis-configured maintainer scripts is still missing.

In this section we devise a classification of possible failures which can occur during upgrade scenarios. We highlight those which are currently most difficult to manage since raised by incomplete or incorrect maintainer scripts. In particular, in Section 2.1 we provide a general description of the failure concept. Then we provide a first classification of possible failures which typically occur on real systems (see Section 2.2). After this general classification, in Section 2.3 examples of recurrent failures are also provided. Finally, we conclude the chapter by discussing how current meta-installers deal with the presented failures in order to precisely identify what are the situations which are still not sufficiently supported.

2.1 General Description of a Failure

The upgrade of a FOSS system typically requires the following steps:

1. a user selects the package he/she aims to install or uninstall;

2. suitable dependency resolution algorithms compute additional packages that are involved in the upgrade;

3. the meta-installer then decides which order to install or remove the files using a sorting algorithm dependent on the meta-installer used;

4. package files are downloaded from the repositories to the local machine;
5. pre installation or removal (for uninstalls) scripts are executed to perform the upgrade.
6. packages are extracted and the files moved to the required locations;
7. post installation or removal (for uninstalls) scripts are executed to perform the upgrade.

If the required packages are available and the dependencies resolution algorithms resolve each dependency, then step *7* can be performed. At this point there are two different types of upgrade failures that can occur: (i) a script may fail during the upgrade sometimes issuing an error-code, and (ii) scripts do not fail but a “not valid” configuration is reached.

Current package management systems can detect script failures only at run-time since they rely on physical resources that are not considered by meta-installers. Meta-installers currently do not analyse the maintainer scripts except for their output. In this setting, problems can be caused by accesses to non-existent files, problems with access rights of files, available memory, bandwidth, databases that are locked or inaccessible, etc. These types of failures will be defined as dynamic failures that are easily rectifiable and reversible because the lists of files associated with a package are known and can be removed or downloaded again. They do not fail as a result of a configuration setting on the host machine and as such can be fixed by current meta-installers.

Focusing on (ii), it is important to understand the notion of “valid” configuration in this context. In general terms a valid configuration is a configuration that does not contain inconsistencies. Of course this is a vague definition, but a better definition can be provided only by defining exactly the different inconsistencies. However, inconsistencies are discovered through the knowledge and experience of failures. The situation, however, is made more complicated by the fact that sometimes errors can only be noticed after some days, weeks and in some cases months. The problem with this type of configuration failure is that the errors will only come to light when a user that experiences an error submits the feedback. Those who are knowledgeable and have enough skills to fix the errors, might not submit their fixes to the maintainers and the fixes might exist only in a decentralised location such as a forum. Those with little experience may receive the error but do not know how to submit an error report. For these reasons and many others, the feedback loop if existent, relies on human intervention and as such can take a long time before the error is resolved. It is expected that the more people that use a particular package, the more testing and people there are that can potentially fix the problem if it arises but this is not always the case. An automated mechanism for detecting these failures would hopefully decrease the rate at which end users are affected by the failures by detecting them and requesting package maintainers produce a patch.

A valid configuration will be defined in this context as a set of inputs into to the simulator such that the simulator produces a result that passes the inbuilt tests and exits the failure-detector with no errors. This maps into computational complexity theory as a “decidable” problem. The proof that the *installability* problem can be encoded as a SAT problem is located in [MBdC+06]. The satisfiability problem is the first example of an NP-complete problem and as such the complexity of satisfying any configuration will be at least NP-complete [Coo71]. Any other configuration will produce a result of a “not valid” configuration as the simulator will not have produced a “valid” result. If the set of all configurations is E then $E = \{V, \neg V\}$ where V is the set of configurations where the simulator produces a “valid” result. $\neg V$ is the logical inverse of the set V. If $\exists V : V \neq \emptyset$ then the “installer” will have to choose a specific V to use. Only notable exceptions such as *apt-pbo*[1] try to calculate an "optimal" solution by using a combination of user choices and algorithms. An optimal V has yet to be defined but it could be in terms of least downloads, least packages upgraded or other cost functions. The first valid V at this moment in time is used in current meta-installers. Such a definition is important since it underpins the upgrade simulation which has two possible outputs: "valid" and "not valid".

A package failure is thus deemed to have occurred whenever a configuration script has reported back an error code (trivial case already detected) or an inconsistent configuration has been reached. Some failure types will be detectable by analysis of the configuration scripts whereas others will only be detectable by using a logging mechanism and noting when a change has not been or could not be reverted. There also might be times when a failure is caused by a modification of the system extraneously from the meta-installer. Depending on the mechanisms used to track the changes in configuration there might be a way of detecting the change(s) and reverting them. It is also dependent on the granularity and the length of the time interval in which changes are made as to whether or not they will be detected.

Therefore as we appreciate that there will be changes to what we can and cannot detect, we follow an iterative and open approach, by adding new inconsistencies as we become aware of them and new techniques are developed for discovering and identifying failures. In the following section we highlight a first failure classification that has been identified through prior research.

2.2 Failure Classification

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

1. *Failure before the package commences installation*, upgrades can fail before the real installation of packages. Hence the system configuration is not changed and this kind of failure can be raised because of the following situations which can occur both singularly and in combination:

   - The package list either locally or remotely has errors;
   - Package management database is locked or inaccessible;
   - Package management database is corrupted, incoherent or is in an erroneous state;
   - Dependencies may be missing due to a broken or unsynchronised repository;
   - Package may refer to a package that has been deprecated and/or removed.

2. *Failure of scripts during upgrades*, these kind of failures rely on the state of physical resources and for this reason it is difficult to simulate or predict. The main causes of these failures can be summarised as follows:

   - Scripts may try to access to non-existent files;
   - Scripts may have problems with access rights to/from files;
   - Scripts referred to may fail e.g. APT-Lua;
   - Pre and Post Install scripts may have insufficient permissions;
   - Syscall or other system failures due to the system missing applications e.g. BASH/Perl;

[1] `http://aptpbo.caixamagica.pt`

  - A file present in the package may already have been installed by another package and be of a different version or be erroneous;
  - Files in one package might touch files from another package. Once this collision has occurred it is difficult to know which version to use and maintain;
  - Package management database may be corrupted after installing to the file system;
  - Scripts could be unable to commit changes to the local package database;
  - A script may refer to a variable or part of the filesystem that has not been defined yet but that would be defined by another package to be installed later in the transaction set. This is different to the other types of script failures in that the same things can occur but will only happen dependant on the order in which the packages are installed;
  - External interruption:
    - Power or I/O failure;
    - Kernel or system fault;
    - User may interrupt the installation especially if it seems that the install has hung. e.g. apt-get install a package when synaptic is running will appear to hang because it has not gained an exclusive lock on the package database. Synaptic on the other hand will check for an exclusive lock before starting.;
  - Insufficient resources, for example disk or memory may become full;
  - when scripts have been created they make assumptions about the state of the system and as such are not idempotent. For instance the creation of a new user could have a guard in which the system checks to see if the user exists before adding the new user.

3. *Non valid configurations are reached*, the upgrade process reaches the end but the obtained configuration is “not valid” since, for instance, there are configuration files which refer to others which no longer exist in the file system;

4. *Slow failures*, even in this case, if the upgrade process reaches the end there is a chance that the obtained configuration contains failures that might not become apparent until another package is released or that it is in a generally unused part of the package and is “logic bomb”;

In addition to this classification, it is worth mentioning the following failures that typically occur during system upgrades.

Script ordering and use-before-define: another kind of failure that has been identified is when the ordering of the script is implicitly important. The term use-before-define is often used in reverse in the context of programming where most languages require the definition of a variable before it can be used (e.g. define before use). The term is being used here to describe the situation where a package that refers to a particular resource is using it for a value but that the value has not been assigned yet. This type of error does not incorporate that of a missing resource that the system should have but rather it is for the more specific case when a value would be defined by another package in the manifest but that the package that would enable the functionality is installed later in the sequence. Sometimes maintainer scripts will exit with a code if there has been this sort of failure, sometimes they will exhibit strange behaviour and sometimes they might use a fall-back mechanism that ultimately leads to a configuration problem. Re-ordering the normally lexically sorted list of packages can highlight problems that are never met in most contexts, because the package maintainer has fixed the packages in such a way that it works with most systems. In RPM-based meta-installers that uses `librpm` for example the process is as follows: *(i)* firstly an rpm transaction is created, *(ii)* next all the packages that the meta-installer wants to install or remove are added to the transaction set, *(iii)* configuration flags are then set for the transaction set and then rpmtsOrder() function is called. If the order returns a non-zero value or if there are any other errors then the transaction will fail, otherwise it will run.

This mechanism does not guarantee if another type of installer is used or if the algorithm used (which is based on tsort style of algorithm [2]) for ordering the packages is changed that the packages will install correctly. It is negatively stable in that it will work as long as the current conditions last but as soon as the algorithm is varied then it will fail. A method for indicating this type of failure is to reverse or randomise the manifest that is provided to the meta-installer. APT does have a mechanism to counteract this which is in the form of another piece of meta-data, `pre-depends`. Other package managers may have similar mechanisms but it is not ubiquitous. The helper utilities do not help to ascertain the correct ordering in which package installation should occur. This has been highlighted in a few locations [3]. These failures, if found through randomisation of the order of the manifests, are normally indicative of a missing pre-dependency between packages. It may be the case though that a dependency cannot be added as it would bring in a dependency loop where two packages inter-rely on each other and can cause the selection algorithms to behave incorrectly.

Multiple provides: one of the other failures that has been identified is that of the multiple provides type of failure. If many packages have the same provides, it is up to the meta-installer to decide which package that best matches the provide. Some "provides" might work as they also have a maintainer-script that sets up some variables that are implicitly depended by the calling package. This could be abstracted as a type of missing dependency but the package maintainer might not think that the other package is a dependency but it needs a more specific case of provides. This again is a type of implicit dependency.

Currently the most intelligent "Package management systems" or meta-installers rely on the maintainers to find dependency requirements and conflicts. Certain helper applications may be able to detect a lot of the possible conflicts and requirements but it is normally based on checks made on the package maintainer's system. Once these dependencies have been calculated they are then static and will not update unless the package is re-released or updated.

All the failures previously described can occur on real systems both in isolation and in combination. In fact, upgrade processes can lead to unrecognised states which can be reached through a mixture of the above or a different unforseen issue. In the rest of the section concrete examples of possible failures will be provided by taking into account the classification previously described.

2.3 Examples of upgrade failures

By taking into account the classification previously presented, in this section we will discuss some concrete examples of upgrade failures. We will discuss how current approaches are able to deal with them and what are the features which are still missing for having comprehensive support of upgrade failures. We will pay particular attention to those failures which depend on fallacious or uncompleted maintainer scripts whose execution lead to "not valid" configurations.

Example 1 Two or more developers may be working on software packages that will when released conflict with the others. As shown in Figure 2.1 Developers of software B and C based on A may check for conflicts with every other package in the package universe. This is already a massive undertaking if taken in a non-automated fashion. If B and C are released and have no knowledge of each other before they are released there is no possibility for B and C to be checked against each other for inconsistencies and to add meta-data to the packages. Certain helper applications may be able to detect a lot of the possible conflicts and requirements but it is normally based on checks made on the package maintainers system. Once these dependencies have been calculated they are then static and will not update unless the package is re-released or updated. Having a mechanism that updates itself with the knowledge of the

[2] `http://rpm.org/api/4.4.2.2/tsort.html`

[3] `http://rpm5.org/docs/api/depends_8c-source.html`

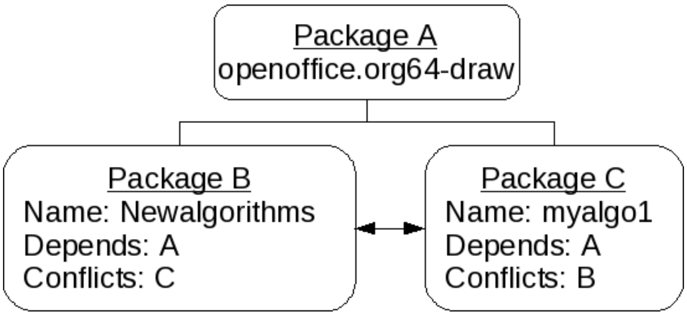


Figure 2.1: Packages with conflicts that are released at a similar time

current set of packages would mean a lot of these potential errors could be avoided. Although software and updates are released continuously the updates performed by the end user and the distribution upgrades tend to happen a lot less frequently. Packages tend to get upgraded at discrete times or at the user/administrator's discretion and so it is possible that the conflicts can be tested before release but not all permutations can be checked and a DSL would help highlight sources of potential configuration errors;

Example 2 An example of a use-before-defines type of failure that leads to an inconsistent system configuration is that which occurs when installing `freeradius`. More specifically in this example, the failure relates to the usage of user and group identification values before they have been defined. There are many other processes related to scriptlet behavior that highlights the general failure type but for this example we use a sub-set failure type that will be exhibited by many packages not just `freeradius`.

The general failure of using an undefined user and group ID (UID and GID respectively) has been approached using different methods. To avoid several different instantiation mechanisms, RPM for instance always uses `getpwnam` and `getgrnam` for resolving the IDs. The issue is that `getpwnam` and `getgrnam` can only be installed once `/etc/passwd` and `/etc/group` exist on the system. This means that packages that are installed before normally use the fallback option of using UID and GID of root. Intervention is needed at this point to make sure the correct UID and GID are assigned to and used by the various packages that have been added to the system. In terms of an example of this type of failure we use that which occurs when installing `freeradius-libs` and `freeradius`. If the library is installed before the `freeradius` package then it would expect a user to be present that is only created by the configuration file in the freeradius package. Although the package maintainer has correctly identified that `freeradius` depends on `freeradius-libs` the package maintainer cannot then add a dependency from `freeradius-libs` back without creating a nested loop. If this is the case some meta-installers that cannot resolve the circular dependency would fail. The problem in this case is that a username is checked for and added as necessary by only `freeradius` package as shown in listing 2.1.

Listing 2.1: Part of the .spec file in freeradius that checks and adds a user, radiusd

```
%pre
getent group  radiusd >/dev/null || /usr/sbin/groupadd -r -g 95 radiusd
getent passwd radiusd >/dev/null || /usr/sbin/useradd  -r -g radiusd -u 95 -c "radiusd user" -s /sbin/nologin radiusd > /dev/null 2>&1
exit 0
```

In the above example `getent` is the UNIX utility for getting entities from the administrative databases such as group or passwd. The database it is requested to search is the first parameter and the second is they key value for which to perform the search. When the database finds an entry marked with the same key it then returns the associated information. In listing 2.1 the group and user radiusd is searched for in the respective administration databases. If an entry is found it will return a non-zero value and the output will be discarded, e.g. passed to `/dev/null`. If the value is found the script will not run the second part however if it cannot be found then the second element in each part of the script following the logical "OR" (||) will be run. The groupadd command is fairly trivial and associates the new radiusd group with Group ID (GID) 95. As for the useradd command there are a number of values that describe how the user fits into the system. The main thing though is that a user is added with the GID defined

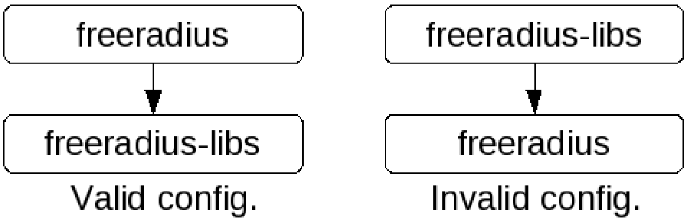


Figure 2.2: Invalid configuration is reached when installing freeradius-libs before freeradius

to be equal to the radiusd grouping. At the end of this script, as long as it completes successfuly we can safely assume that a user and group called radiusd exists. It can then be used for reference, like by `freeradius-libs`.

If the non-library package installs first then it is fine, otherwise it will fail. The example can be found in the CentOS mailing lists [4]. Although some package management systems have mechanisms for avoiding this type of failure by creating additional meta-data, there is a lack of enforcement of this by any of the existing helper tools. Serialization of side-effects of scriplet actions can help catch these flaws. It is a particularly interesting error in that depending on changes in the algorithm it may not be reproducible on another system given the same inputs and actions carried out by the end user. Only those who fully understand what the meta-installer is doing might be able to catch this type of failure through looking at the logs and recognizing that the order of package installation is leading to the invalid configuration state.

Example 3 Although not a failure per-se the following example highlights a deficiency of current meta-installers. Many maintainer scripts run cache rebuilders to make sure that any new changes are integrated into the system. They are normally added into maintainer scripts by package maintainers to make sure that (de-)associations are made at the correct time. The issue is that many package maintainers now use the cache-rebuilders whether or not they are needed. Many maintainer scripts are created by helper utilities but of those that are hand-written, from the analysis performed, it would appear that a significant number are copied and modified. It has now become the norm to run cache-rebuilders and in some cases it is required to pass validation tests, for instance it is mandated by the Debian Policy. The processing power of modern systems means that running these cache-rebuilders does not affect too much on performance. However with large upgrade procedures such as a distribution upgrade the small performance hit is repeated many times by many packages and can slow the package installation process. What was worse was if the cache had to be rebuilt several times if there was a shortage of memory available to the cache [5], however these issues have had various patches at different stages and now since `glibc 2.7` the patch is incorporated. By using a model of the system that has a knowledge of the libraries installed as one of the meta-classes, it is possible for the simulator to only run `ldconfig` when necessary and optimise the number of times the cache rebuilder is run. Calculating the minimal/optimal number of times a cache-rebuilder has to be run is of course an additional process that has its own limitations. What can be done though is to analyse the maintainer scripts sequentially as each package is installed to see if a library has been installed since the previous configuration and if it is to be used by an executable file then run the cache rebuilder. This will not optimise the number of rebuilds performed for the whole set of packages but will minimise the number of times that a cache rebuild is performed on the set in that particular order. By removing the redundant cache rebuilds we can highlight one of the benefits of the DSL that is not ubiquitous amongst standard meta-installers at the moment.

Example 4 An example of possible upgrade which can lead to a non valid configuration is presented in the following by considering `Apache2` and `PHP5` which is a scripting language integrated with the Apache Web server. In particular, let us assume that we want to remove the package `libapache-mod-php5` from the filesystem. Then the `PHP5` module in the Apache configuration has to be disabled before its removal. This is necessary, otherwise inconsistent configurations can be reached like the one shown

[4] `http://lists.centos.org/pipermail/centos-devel/2009-March/004250.html`

[5] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=431143`

in Figure 2.3. The figure reports the sample `Configuration2` which has been reached by removing `libapache-mod-php5` without changing the configuration of `apache2`. Such a configuration is broken since it contains a dependency between the `apache2` and `libapache-mod-php5` package settings, when only `apache2` is installed. If Apache is now run as a service it may refer to the missing package, `PHP5` which of course could lead to many errors including security breaches and services crashing.

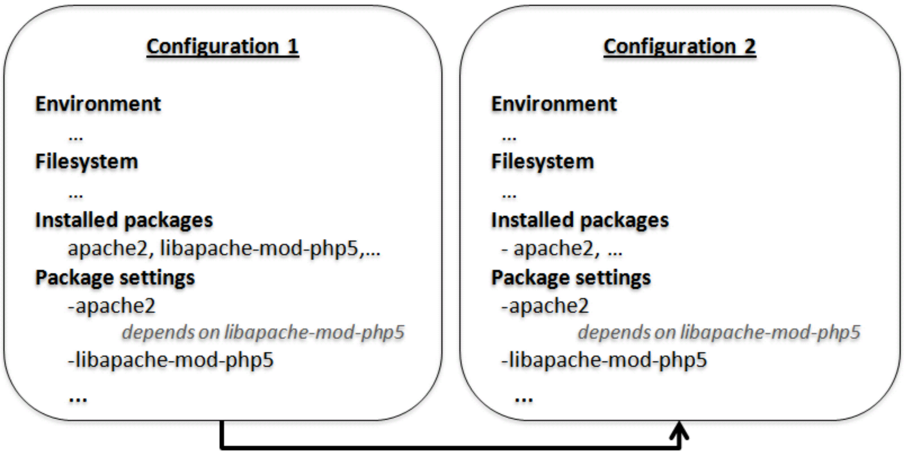


Figure 2.3: Incorrect package removal

In order not to lead to invalid configurations like the one in Figure 2.3, the package `libapache-mod-php5` contains the *postinst* and *prerm* scripts reported on the left hand-side and right hand-side in the following snippet, respectively.

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

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

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

It is important to note that currently, available package managers are not able to predict inconsistencies which can occur if maintainer scripts are not complete. For instance, the invalid configuration reported in Figure 2.3 can be obtained if the *prerm* above does not contain the statement '`a2dismod php5`'. Package managers are not able to detect that such a statement is missing or that an invalid configuration has been reached. The only types of errors they can detect are run-time failures of scripts. They cannot normally resolve the problem and perform a limited set of actions at this point. They may try and restart the configuration script, report the error to the user or more frequently than not they will abort the script and possibly provide an error code and/or warning message.

2.4 How Meta-Installers deal with upgrade failures

Currently the system for how meta-installers cope with failures is distribution and meta-installer dependent. Most will use algorithms as identified earlier to detect if a solution to the satisfiability problem exists and generate a package transaction set to install.

Once they have a set of packages to install the order in which they are installed again depends on the algorithms used but as examined before there are a whole class of problems that can be identified by

	Existing approaches	**Model driven approach**
Before upgrade	Y	Y
During upgrade	Y[6]	Y
After upgrade	N	Y
Slow failures	N	Y

Table 2.1: Current support to *detect* and *manage* upgrade failures

randomising the order in which packages are installed. If there are no circular dependencies identified that cannot be resolved, i.e. for `rpm`, `rpmtsOrder()` would come back with a value of zero then there exists a solution set for the packages to install. If all these pre-conditions are met and an ordered list of packages has been stored into a transaction set the meta-installers at this stage act as dumb-agents and call `dpkg`, `rpm` or the associated installer on the packages and report the error codes at this stage. The installer will then unpack the files downloaded by the meta-installer and run the associated configuration scripts. It is at this stage that the meta-installers assume that the configuration files will report an error code if something has gone wrong.

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

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

To summarize, Table 2.1 reports the main categories of possible upgrade failures that we identified and described in the previous sections. For each of them the table states if current approaches are able or not to *detect* and eventually *manage* them. The model driven approach presented in the deliverable D2.1 and recalled in the next chapter has been conceived to focus on detecting failures before, during and after upgrades are performed. The DSL plays a key role in the overall approach since the upgrade simulation and the failure detection rely on the maintainer script behavior and on configuration models which abstract the real systems.

Chapter 3

Model-driven approach for supporting FOSS system upgrades

The problem of maintaining FOSS installations is far from trivial and has not yet been addressed properly [DTZ08]. In particular, current package managers are neither able to predict nor to counter vast classes of upgrade failures. The main reason is that package managers rely on package meta-information only (in particular on inter-package relationships), which are not expressive enough. In the deliverable D2.1 and in [CRP$^+$09] we proposed an approach consisting of maintaining a model-based description of the system and simulating upgrades in advance on top of it, to detect predictable upgrade failures and notify the user before the actual installation occurs and 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 missing sub-components), which are currently not expressible as inter-package relationships.

In this chapter we recall the phases of FOSS upgrades (see Section 3.1) and outline the concepts of Model Driven Engineering (MDE) (see Section 3.2). Then the model-driven approach already proposed in D2.1 is summarized in Section 3.3 even though we focus more on the role of the DSL which has been conceived to specify the behavior of the maintainer scripts and to predict several of their effects on package upgrades (see Section 3.4).

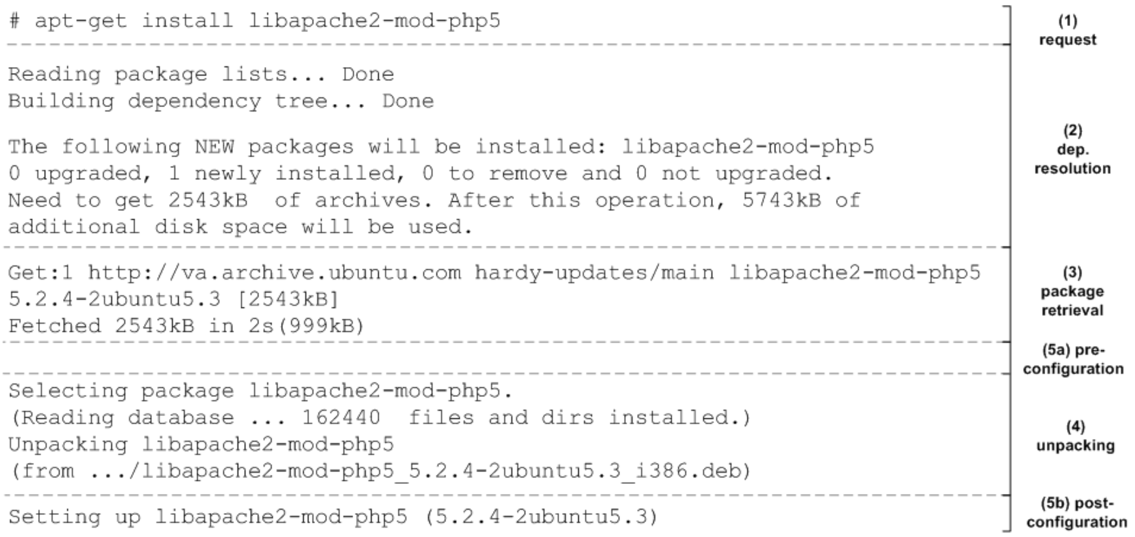
3.1 FOSS system upgrades

The different phases of the so called *upgrade scenario* are summarized in Table 3.1, using as an example the popular `APT` meta-installer. The process starts in phase (1) with the user requesting to alter the local package status. The expressiveness of the requests varies with the meta-installer, but the aforementioned actions (install, remove, etc.) are ubiquitously supported, possibly with different semantics [TZ08].

Phase (2) checks whether a package satisfying the dependencies and conflicts exists (the satisfiability problem is at least NP-complete [EDO06]). If this is the case one is chosen in this phase. Usually the one that occurs first is chosen but different tools now allow the user to specify what constraints should be upheld (eg. see apt-pbo[1] by Caixa Mágica Linux). 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).

Intertwined with package retrieval and unpacking, there can be several configuration phases, (exemplified by phases (5a) and (5b) in Table 3.1), where maintainer scripts get executed. The details depend on the available hooks; `dpkg` offers: pre/post-installation, pre/post-removal, and upgrade to some version [JS08].

[1] `http://aptpbo.caixamagica.pt`



```
# apt-get install libapache2-mod-php5
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.
Get:1 http://va.archive.ubuntu.com hardy-updates/main libapache2-mod-php5
5.2.4-2ubuntu5.3 [2543kB]
Fetched 2543kB in 2s(999kB)
Selecting package libapache2-mod-php5.
(Reading database ... 162440  files and dirs installed.)
Unpacking libapache2-mod-php5
(from .../libapache2-mod-php5_5.2.4-2ubuntu5.3_i386.deb)
Setting up libapache2-mod-php5 (5.2.4-2ubuntu5.3)
```

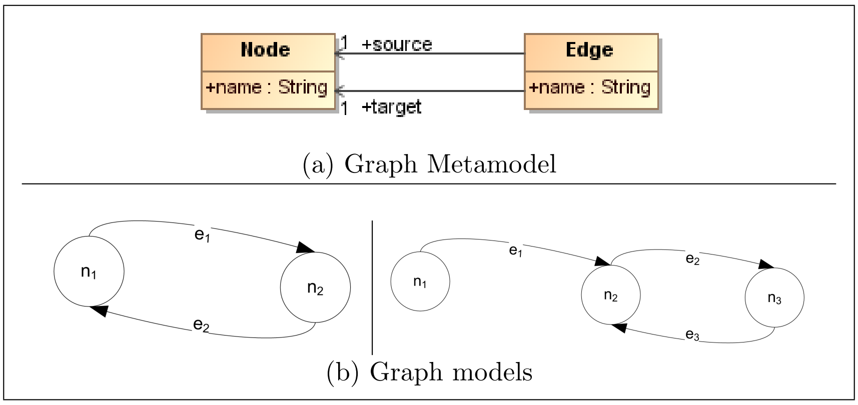
Table 3.1: The package upgrade process

Each phase of the upgrade process can fail as well as package deployment. Trivial failures, for example, network or I/O failures before configuration, can be easily dealt with when considered in isolation: the whole upgrade process can be aborted and the unpack stage can be undone, since all the involved files are known; they can be removed safely and no upgrade is performed and thus the system is unchanged. Maintainer script failures can not be as easily undone, nor prevented. Scripts are implemented in Turing-complete languages, and all non-trivial properties about them are undecidable, including determining a-priori their effects to be able to revert them upon failure [DTZ08]. In this respect, state of the art package managers do not provide support to simulate system upgrades taking the behavior of maintainer scripts into account. In fact, current tools consider only inter-package relationships which are not enough to predict side-effects and system inconsistencies which can be encountered during upgrades. They also often rely on the competance of the package maintainer. If a standard template or assistance tool is used, these types of errors are minimised but often the experience of the package maintainer is relied upon to know whether their package will conflict with any others. As the number of packages available tends to increase the maintainer also has to think in advance of what type of packages might conflict but have not been developed yet. This of course is not feasible and increases the chances of a failure due to the lack of expressiveness of or the insufficient meta-data.

3.2 Model Driven Engineering

MDE [Sch06] 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 modelling 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.

MDE relies on a conceptual framework consisting of *model*, *meta-model*, and *model transformation* which are described in the rest of the section.



(a) Graph Metamodel

(b) Graph models

Figure 3.1: Models conforming to a sample metamodel

3.2.1 Models and Meta-models

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.

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-modelling* 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-modelling can be seen as the construction of a collection of “concepts” (things, terms, etc.) within a certain 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 [B0́5]. For instance, Fig. 3.1.a depicts a sample meta-model containing the concepts and the relations between the elements of a graph. 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 the graph meta-models previously mentioned are reported.

Object Management Group (OMG) has introduced the four-level architecture illustrated in Figure 3.2. At the bottom level, the `M0` layer is the real system. A model represents this system at level `M1`.

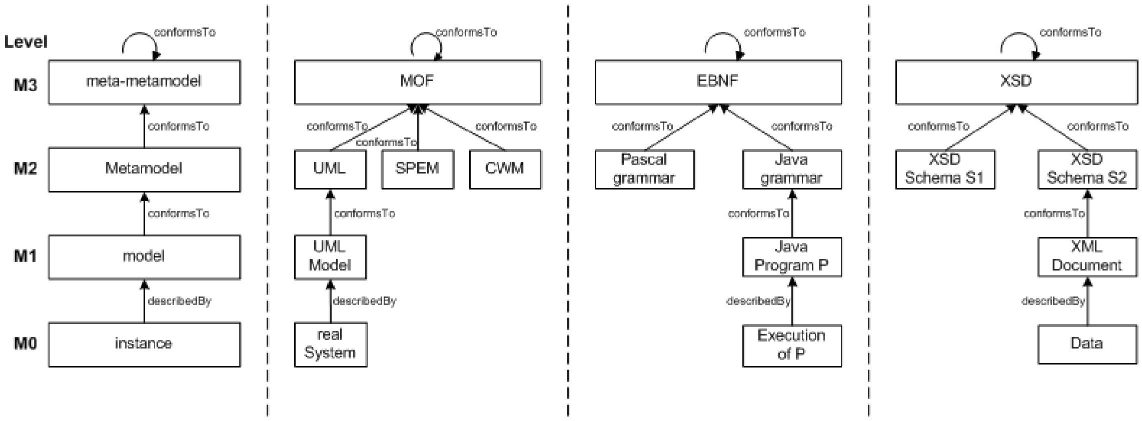


Figure 3.2: The four layers meta-modelling architecture

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) [Obj03a] as a standard for specifying meta-models. For example, the Unified Modeling Language (UML) meta-model [Obj03b] is defined in terms of MOF. A supporting standard of MOF is **XMI!** (**XMI!**) [Obj03c], which defines an XML-based exchange format for models on the `M3`, `M2`, or `M1` layer. This meta-modelling architecture is common to other technological spaces as discussed by Kurtev et al. in [AKB02]. For example, the organisation 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 meta-modelling, *model transformation* is also a central operation in MDE as discussed in the next section.

3.2.2 Model Transformations

In addition to meta-modelling, *model transformation* is also a central operation in MDE. The MDA guide [OMG03] 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.

These aspects will be discussed in more detail in Chapter 4.3, which uses model transformations for specifying the semantics of the proposed DSL for specifying maintainer scripts.

3.3 Simulating system upgrades

The model-driven approach depicted in Figure 3.3 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 their effects on the operation of software distributiona. To this end, models can be used to drive rollback operations, to recover previous configurations according to user decisions or after upgrade failures. Being more precise, the simulation takes two models as input: the *System Model* and the *Package Model* (see 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. One of the most important aspects of the simulation is the behaviour of the maintainer scripts that are defined using the DSL which is presented in depth in the next chapter. Taking this further, given the current configuration, defined in terms of a system model, the simulator simulates upgrading the system by taking into account the packages which have to be installed and/or removed. In particular, for each package involved in the package model, the associated maintainer scripts are executed. If the execution of a script has some problems or a *'not valid'* configuration is reached, then the outcome of the simulation will be *'not valid'* (see arrow ⓒ). In this case it is taken for granted that the upgrade on the real system will likely fail. Thus, before proceeding with upgrading the problem(s) identified by the simulation should be fixed. Conversely, if the overall upgrade leads to a new configuration that is a *'valid'* configuration, then the simulation outcome will be *'valid'* (see arrow ⓓ). In this case, the upgrade on the real system can be performed (see arrow ⓘ). However, since the models are an abstraction and hence a simplification of the real system as discussed in Section 3.2.1, upgrade failures might still occur. In the scenario that our simulation approach is unable to detect failures the system will capture the logs but the original upgrade would take place just as if simulation was not adopted.

In order to support the down-grade, during the real upgrade of the system, the models are continuously updated and kept aligned with the real system that they are modelling. Furthermore, *Log models* are

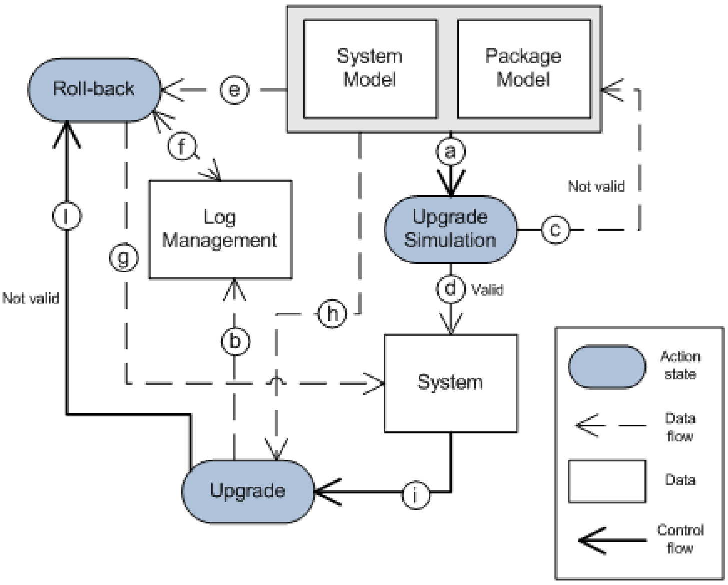


Figure 3.3: Overall approach

produced in order to store all the transitions between configurations (see arrow ⓑ). The information contained in the system, package, and log models (arrows ⓔ and ⓕ) are used in case of failures (arrow ⓛ) or in the case that a user decides to perform a down-grade at a later stage. In other words, models drive the down-grade, by indicating the reverse actions that must be performed to bring the system back to a previous valid configuration (arrow ⓖ). This model-driven approach, to be effective, must be integrated with other approaches able to physically store useful system information, and be able to retrieve these pieces of information as needed. A reliable transaction store of the logs is required as any corruption on the store will otherwise mean that all the configuration states are no longer useful.

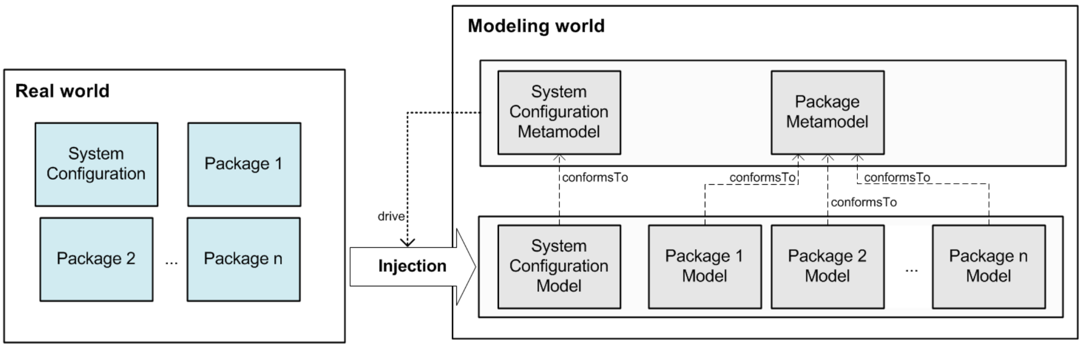


Figure 3.4: Model injection

In order to apply the approach depicted in Figure 3.3 onto a real scenario, existing systems have to be represented in terms of models. In this respect, the availability of *injectors* is crucial since they are tools that transform software artifacts into corresponding models in an automatic way. In particular, as shown in Figure 3.4, given a real software system and a set of packages a corresponding representation in the modelling world has to be obtained. 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. In this respect, models are specified by using modelling constructs which are formalized in specific metamodels that have been conceived during a domain analysis phase (see the deliverable D2.1).

Over the last years, several approaches for extracting models from software artifacts have been proposed even though the optimal solution which can be used for any situation does not exist yet [JJJ08]. The complexity of the problem relies on the limitation of current lexical tools which do not provide the proper

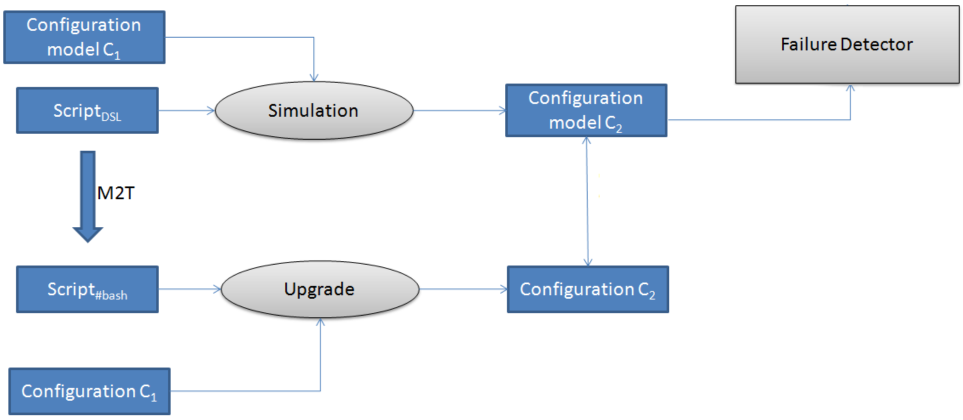


Figure 3.5: Role of the DSL in the upgrade

abstractions and constructs to query code and generate models with respect to given metamodels. Some approaches like [WK06, Eff06] focus on generating metamodels from grammars but they have some drawbacks that may restrict their usefulness, such as the poor quality of the automatically generated metamodel [JJJ08]. Approaches like [JBK06] enable the automatic generation of injectors starting from annotated metamodels with syntactic properties. However, they do not permit the reuse of existing grammars written for well-known parser generators. Techniques like [JJJ08] propose specific languages to query software artifacts and generate models according to specified source-to-model transformation rules.

Several projects are under development to provide tools and methodologies for model-driven modernisation and model injection. For instance, MoDisco [Ecl] defines an infrastructure for supporting model-driven reverse engineering by relying on the concept of *discoverer* which is a piece of software in charge of analysing part of an existing system and extracting a model using the MoDisco's infrastructure. However, the automatic representation of real systems in terms of models is the main topic of the deliverable D2.2. In this respect, in the rest of this document, we will assume the availability of models without taking into account how we have obtained them.

In the next section, the role of the DSL in the simulation of system upgrades is explained in more detail.

3.4 Role of the DSL in the upgrade scenario

The rationale behind a brand new language for specifying maintainer scripts is forcing distribution maintainers to use a subset of high level statements (a sort of macro) with well defined semantics. The well defined semantics enables a deep understanding of how the system evolves into a new configuration by executing the single statement. Furthermore the DSL is not Turing-complete because by definition the DSL was designed to describe an abstraction of systems and to focus on the main aspects of the particular problem domain. By adding Turing-complete elements we increase the expressive power of the language but we lose the benefits that we gain by having that abstraction. It is a consideration that has been taken into account when designing and specifying the DSL.

As depicted in Figure 3.5 the simulation of a package installation relies on an abstract representation of the source configuration and executes the maintainer scripts whose behavior is specified by means of the proposed DSL. The aim of the simulation is to identify possible inconsistencies which can be detected during the maintainer scripts execution or once a target configuration is obtained. For instance, the execution of a statement on a service which does not exist in the current configuration can stop the simulation. In this way the maintainer can identify and modify the script in order to solve such a problem. Then the *failure detector* is used to analyze target configurations and check if inconsistencies occur. In the case of "valid" configuration, the installation is performed on the real system. In particular, the scripts specified with the DSL are transformed in to target code written in currently used, Turing complete, scripting languages. The scripts obtained are then executed on the real configuration.

To summarize, the DSL will have two different implementations each of them at different level of abstractions. A first implementation is represented by a simulator which is able to simulate package upgrades and check if they lead systems to inconsistent states or not. Another implementation is represented by a metainstaller which will be able to execute DSL scripts on real systems. In the rest of the document we will focus on the DSL by presenting its building concepts, and semantics without referring to particular implementations. In fact, the support of the DSL and its application on real scenarios is the focus of forthcoming deliverables, like D2.3 and D3.3.

Chapter 4

DSLs supporting the upgradeability of GNU/Linux systems

DSLs are languages able to raise the level of abstraction beyond coding by specifying programs using domain concepts [TK05]. In particular, by means of DSLs, the development of systems can be realized by considering only abstractions and knowledge from the domain of interest. This contrasts with General Purpose Language (GPL), like C++ or Java, that are supposed to be applied for much more generic tasks in multiple application domains. By using a DSL the designer does not have to be aware of implementation intricacies, which are distant from the concepts of the system being implemented and the domain the system acts in. Furthermore, operations like debugging or verification can be entirely performed within the domain boundaries.

Over the years, many DSL have been introduced in different application domains (telecommunications, multimedia, databases, software architectures, Web management, etc.), each proposing constructs and concepts familiar to experts and professionals working in those domains.

As any other computer language (including GPLs), a DSL consists of abstract and concrete syntax definition and possibly a semantics definition, which may be formulated at various degrees of preciseness and formality. In the context of MDE we conceive of the DSL as a collection of coordinated models. We are in this way, leveraging the unification power of models [Béz05]. Each of the models composing a DSL specifies one of the following language aspects:

- *Abstract syntax.* As we discussed before, the basic distinction between DSLs and GPLs is based on the relation to a given domain. DSLs have a clearly identified, concrete problem domain. Programs (sentences) in a DSL represent concrete states of affairs in this domain. A conceptualization of the domain is an abstract entity that captures the commonalities among the possible state of affairs. It introduces the basic abstractions of the domain and their mutual relations. Once such an abstract entity is explicitly represented as a model it becomes a metamodel for the models expressed in the DSL. It plays a central role in the definition of the DSL. For example, a DSL for directed graph manipulation will contain the concepts of nodes and edges, and will state that an edge may connect a source node to a target one. Similarly, a DSL for Petri nets will contain the concepts of places, transitions and arcs. Furthermore, the metamodel should state that arcs are only between places and transitions;
- *Concrete syntaxes.* A DSL may have different concrete syntaxes. For instance, a possible concrete syntax of a Petri net DSL may be defined by mapping places to circles, transitions to rectangles, and arcs to arrows. The display surface metamodel in this case has the concepts of Circle, Rectangle, and Arrow;
- *Dynamic semantics.* Generally, DLSs have different types of semantics. For example, OWL [Wor] is a DSL for defining ontologies. The semantics of OWL is defined in model theoretic terms. The semantics is static, that is, the notion of changes in ontologies happening over time is not captured.

> Many DSLs have a dynamic semantics based on the notion of transitions from state to state that happen in time. Dynamic semantics may be given in multiple ways, for example, by mapping to another DSL having itself a dynamic semantics or even by means of a GPL.

In this chapter, all these aspects of the DSL conceived for the maintainer script specifications will be presented in depth. In particular, Section 4.1 presents the abstract syntax in terms of a refined version of the MANCOOSI metamodels introduced in the deliverable D2.1. The concrete syntax of the language is defined in Section 4.2. Finally, the semantics of the language is given in Section 4.3.

4.1 MANCOOSI DSL: Abstract syntax

In order to identify the right trade-off between model completeness and usefulness we analyzed two complex FOSS distributions: Debian[1], the largest distribution in terms of number of software packages [AGRH05] and RPM-based Fedora[2] distributions. The first result of the analysis has been the definition of the metamodels presented in the following. However, the most challenging part of the conducted analysis was the analysis of maintainer scripts. In fact since our aim is to define the DSL for writing maintainer scripts it is of fundamental importance to known what are the statements that must be included by the DSL. In other words the DSL should contain a restricted and well defined subset of statements, but the DSL must be expressive enough. Thus, our aim is to describe the most common macro-actions of maintainer scripts in terms of models which abstract from the real system, but are expressive enough to grasp several of their effects on package upgrades.

The adopted scripting languages are mainly POSIX shell but they are written also in Perl [The09b], Bash [The09a], etc. 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 (file)system.

Due to the large amount of scripts to consider (e.g., about 25·000 on Debian Lenny), we tried to collect scripts in clusters to be able to concentrate the analysis on representatives of the equivalence classes identified. The adopted procedure for clustering has been presented in the deliverable D2.1 and in [RPPZ09] and is schematised in the following:

1. *Collect all maintainer scripts* of a given distribution;
2. *Identify scripts generated from helper tools.* Since a large part of maintainer scripts are automatically generated using “helper” tools (which provide a collection of small, simple and easily understood tools that are used to automate various common aspects of building a package) we can concentrate the analysis on the helpers themselves, rather than on the result of their usage;
3. *Ignore inert script parts.* We then found and ignored inert script parts, i.e., script parts which do not affect their computational state such as blank lines of comments;
4. *Study of scripts written “by hand”.* We analyzed the remaining scripts tying to identify recurrent templates that maintainers use when writing the scripts.

Summarizing, our analysis of Debian and Fedora highlighted the it is possible to define a higher-level language that can be substituted to current script languages. Thanks to the analysis we are now able to define statements that are necessary for the DSL and, on the other side, statements that should be out of scope of the DSL.

Their analysis has induced the definition of three metamodels (see Figure 4.1) which describe the concepts making up a system configuration and a software package, and how to maintain the log of all upgrades. The metamodels have been defined according to an iterative process consisting of two main steps *a)* elicitation of new concepts from the domain to the metamodel *b)* validation of the formalisation of

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

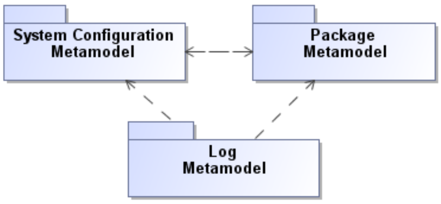


Figure 4.1: Dependencies among metamodels

the concepts by describing part of the real systems. In particular, the analysis has been performed considering the official packages released by the distributions with the aim of identifying elements that must be considered as part of the metamodels. We report here only the results of the analysis, i.e., the metamodels themselves:

- the *System Configuration metamodel*, which contains all the modeling constructs necessary to make the FOSS system able to perform its intended functions. In particular, it specifies installed packages, configuration files, services, filesystem state, loaded modules, shared libraries, running processes, etc. The system configuration metamodel takes into account the possible dependency between the configuration of an installed package and other package configurations. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by the proposed metamodels which embody domain concepts which are not taken into account by current package manager tools;
- the *Package metamodel*, which describes the relevant elements making up a software package. The metamodel also gives the possibility to specify the maintainer script behaviors which are currently ignored—beside mere execution—by existing package managers. In order to describe the scripts behavior, the package metamodel contains the `Statement` metaclass, see Figure. 4.2, that represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration;
- the *Log metamodel*, which is based on the concept of transactions that represent a set of statements that change the system configurations. Transitions can be considered as model transformations [B0́5] which let a configuration C_1 evolve into a configuration C_2.

As depicted in Figure 4.1, *System Configuration* and *Package* metamodels have mutual dependencies, whereas the *Log* metamodel has only direct relations with both *System Configuration* and *Package* metamodels.

The proposed metamodels represents a step toward *(i)* simulation of package installations, *(ii)* fault and roll-back management, *(iii)* log management. Simulation of package installations and fault management are achieved thanks to the *Configuration* and *Package* metamodels. Roll-back management and log management are instead achieved by means of the *Log* metamodel. In fact, roll-back operations can exploit the information contained in the log models which store the transactions between different configurations. In the next section we will focus on the the package metamodel since it embodies the abstract syntax of the DSL which is the main outcome of this document.

4.2 MANCOOSI DSL: Concrete syntax

In this section we define the notation the end user will use to specify scripts conforming to the abstract syntax (defined by means of the Mancoosi metamodel previously outlined). The DSL consists of statements which modify the system configuration that is modeled in terms of File system, Environment and Package setting elements. Moreover, there are **control** and **iterator statements** which can be used to specify the application of the other statements. They will be explained in Section 4.2.2 and 4.2.3, respectively.

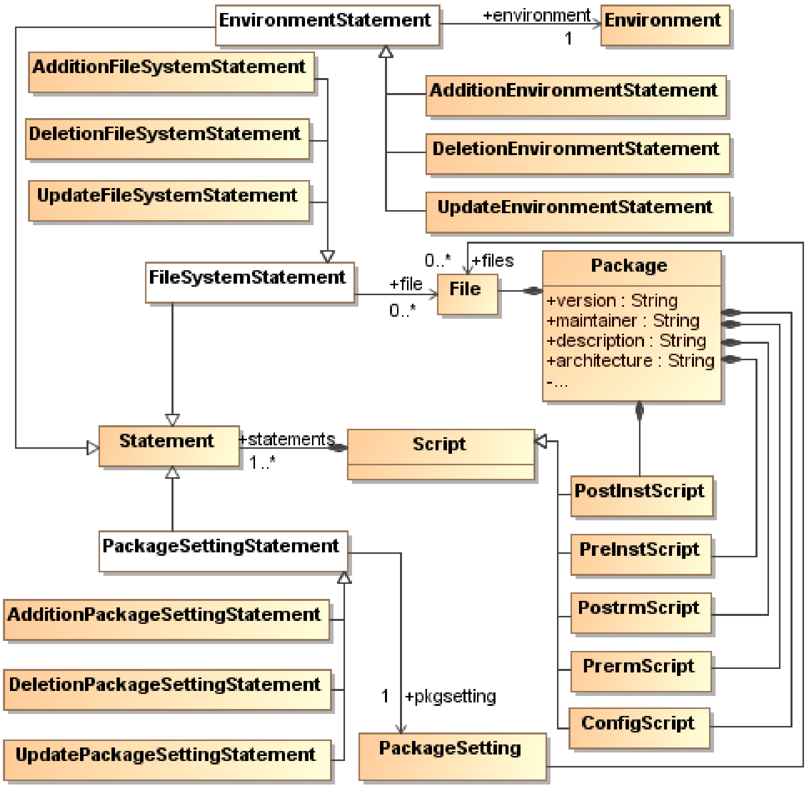


Figure 4.2: Fragment of the Package metamodel

4.2.1 Grammar Definition

The Grammar definition uses Extended Backus-Naur Form as standardised by ISO/IEC 14977. It is a formal mathematical description (metasyntax) that is used to describe a context-free grammar. EBNF varies from BNF in that it allows more operators that define the number of times objects can appear. EBNF can be represented in BNF and vice versa[3]. ε refers to an empty set and so can be used to simplify a syntax instead of using optional operators.

```
SCRIPT ::= STATEMENT_LIST


STATEMENT_LIST ::= TEMPLATE_STATEMENT STATEMENT_LIST |
                   CONTROL_STATEMENT STATEMENT_LIST |
                   ITERATOR_STATEMENT STATEMENT_LIST |
                   TAGGING_STATEMENT STATEMENT_LIST |
                   ε


CONTROL_STATEMENT ::= CASEPOSTINST | CASEPOSTRM | CASEPRERM | CASEPREINST


CASEPOSTINST ::= CasePostinst(){
                 CONFIGURE
                 ABORTUPGRADE
                 ABORTREMOVE
                 ABORTDECONFIGURE
                 }
```

[3] http://www.garshol.priv.no/download/text/bnf.html

```
CONFIGURE ::= configure:  STATEMENT_LIST, | ε

ABORTUPGRADE ::= abortUpgrade:  STATEMENT_LIST, | ε

ABORTREMOVE ::= abortRemove:  STATEMENT_LIST, | ε

ABORTDECONFIGURE ::= abortDeconfigure:  STATEMENT_LIST, | ε

CASEPOSTRM ::= CasePostrm(){
               PURGE
               REMOVE
               UPGRADE
               FAILEDUPGRADE
               ABORTINSTALL
               ABORTUPGRADE
               DISAPPEAR
               }

PURGE ::= purge:  STATEMENT_LIST, | ε

REMOVE ::= remove:  STATEMENT_LIST, | ε

UPGRADE ::= upgrade:  STATEMENT_LIST, | ε

FAILEDUPGRADE ::= failedUpgrade:  STATEMENT_LIST, | ε

ABORTINSTALL ::= abortInstall:  STATEMENT_LIST, | ε

ABORTUPGRADE ::= abortUpgrade:  STATEMENT_LIST, | ε

DISAPPEAR ::= disappear:  STATEMENT_LIST, | ε

CASEPRERM ::= CasePrerm(){
              REMOVE
              UPGRADE
              DECONFIGURE
              FAILEDUPGRADE
              }

DECONFIGURE ::= deconfigure:  STATEMENT_LIST, | ε

FAILEDUPGRADE ::= failedUpgrade:  STATEMENT_LIST, | ε

CASEPRERM ::= CasePreinst(){
              INSTALL
              UPGRADE
              ABORTUPGRADE
              }
```

```
INSTALL ::= install:  STATEMENT_LIST, | ε

ITERATOR_STATEMENT ::= ITERATOR_DIRECTORY | ITERATOR_FILE | ITERATOR_ENUMERATION |
                       ITERATOR_PARAMETER | ITERATOR_WORD

ITERATOR_DIRECTORY ::= ForEachFile(DIRECTORY,ORDER){
                       STATEMENT_LIST,
                       }

ITERATOR_FILE ::= ForEachLine(FILE,ORDER){
                  STATEMENT_LIST,
                  }

ITERATOR_ENUMERATION ::= ForEachElement(ENUMERATION,ORDER){
                         STATEMENT_LIST,
                         }

ITERATOR_PARAMETER ::= ForEachArg(PARAMETER,ORDER){
                       STATEMENT_LIST,
                       }

ITERATOR_WORD ::= ForEachChar(STRING,ORDER){
                  STATEMENT_LIST,
                  }

ORDER ::= left | right

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

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

DESKTOP ::= update_desktopdb

DOCBASE ::= inst_doc(STRING) | rm_doc(STRING)

EMACS ::= inst_emacs_package(STRING) | rm_emacs_package(STRING)

GCONF ::= preinst_gconf | postinst_gconf
          postrm_gconf | prerm_gconf

ICONS ::= update_icons

INFO ::= inst_info(STRING) | rm_info(STRING)
```

```
INIT ::= postinst_init(STRING) | postinst_init_nostart(STRING)
         postinst_init_restart(STRING) | prerm_init(STRING)
         postrm_init(STRING)

LIBRARY ::= update_libs

MENU ::= update_menu | postinst_menu | postrm_menu

MIME ::= update_mime(STRING)

MODULES ::= update_modules(STRING)

SCROLLKEEPER ::= update_scrollkeeper

SGML ::= postinst_sgmlcatalog | prerm_sgmlcatalog | postrm_sgmlcatalog

UDEV ::= preinst_udev | postinst_udev

USRLOCAL ::= postinst_usrlocal | prerm_usrlocal

USERGROUPS ::= user_add(STRING,STRING,STRING) | user_remove(STRING,STRING)
               group_add(STRING) | group_remov(STRING)

WM ::= postinst_wm(STRING,STRING) | postinst_wm_noman(STRING)
       prerm_wm(STRING)

XFONT ::= update_xfonts

TAGGING_STATEMENT ::= add<metaclass name>(<metaclass feature values>)
                      delete<metaclass name>(<metaclass feature values>)
                      add<metaclass name>_<attribute>(<element>,<attribute value>)
                      delete<metaclass name>_<attribute>(<element>)
                      add<metaclass name>_<reference>(<source element>,<target element>)
                      delete<metaclass name>_<reference>(<source element>,<target element>)
```

4.2.2 Control statements

Control statements give an explicit control flow specifying statement executions. Depending on the script in which they are located, they can be in turn classified into:

- **case_postinst**, it is used in maintainer scripts that would normally be triggered after performing the installation of the package. As such it is normally used to register that the installation was successful and allow other applications and sym-links to refer to the newly upgraded/(installed) package. The elements identified that a maintainer script would then utilise are: `configure`, `abortUpgrade`, `abortRemove`, `abortDeconfigure`. A sample use of the `case_postinst` statement is reported in the following:

```
case_postinst{
    configure: statementList ,
    abortUpgrade: statementList ,
    abortRemove: statementList ,
    abortDeconfigure: statementList
}
```

- **case_postrm**, it is used in maintainer script that would be activated after the removal of a package. It is normally used to modify databases of installed packages or to remove referenced links. The possible states that this can have are: `purge`, `remove`, `upgrade`, `failedUpgrade`, `abortInstall`, `abortUpgrade` and `disappear`. A sample use of the `case_postrm` statement is given in the following:

```
case_postrm{
    purge: statementList ,
    remove: statementList ,
    upgrade: statementList ,
    failedUpgrade: statementList ,
    abortInstall: statementList ,
    abortUpgrade: statementList ,
    disappear: statementList
}
```

- **case_prerm**, it will only occur when the parent template has selected that this part of the maintainer script is in the state of just before removing the package. The possible templates that can exist below are identified as: `remove`, `upgradem`, `deconfigure`, `failedUpgrade`. In the following a sample use of the `case_prerm` statement is reported:

```
case_prerm{
    remove: statementList ,
    upgrade: statementList ,
    deconfigure: statementList ,
    failedUpgrade: statementList
}
```

- **case_preinst**, the statement occurs when the case pre-installation has been selected by the parent template. There are three potential states that have been identified that a maintainer script would use before an install and these are: `install`, `upgrade` and `abortUpgrade`.

```
case_preinst{
    install: statementList ,
    upgrade: statementList ,
    abortUpgrade: statementList
}
```

4.2.3 Iterator statements

Iterator statements, depending on the considered collection, can be in turn classified into:

- *Directory iterator*, which permits to iterate on the files contained in a given directory. The **forEachFile** statement takes two arguments: directory and order. The template will then sequentially select each file for use in the direction defined by order which can consist of the keyword `ascending` or `descending`. Since the elements will be lexicographically ordered, the `descending` keyword is used to instruct the iterator which has to consider the elements that appear before in the considered sequence. Viceversa, the `ascending` keyword is used to consider first the elements which appear after in the sequence. This way to specify the order is the same for all the iterators provided by the DSL.

  Each file can be manipulated by further control templates. Once the last file has been reached the template will have no more to access and this control template will return control back to its parent template.

```
forEachFile(directory,order){
    statementList ,
}
```

- *Lines of a file iterator*, to specify iterations on the lines of a given file. In particular, the **forEachLine** statement will take as arguments two parameters file and order. File is the document separated by carriage-return line-feeds. The control template will read each line in the file in the direction specified by order and the template will be able to use the current line for further control. The file will be read sequentially until the end of file is reached (EOF).

```
forEachLine(file,order){
    statementList,
}
```

- *Enumeration iterator*, takes an enumeration iterator to work on an ordered set of index;value pairs in the direction specified by order.

```
forEachElement(enumeration,order){
    statementList,
}
```

- *Input parameter iterator*, to iterate iterate through each argument that has been passed to the script in an order as defined by a parameter of the same name. Order type is not specified but it could be in terms of sequential ordering or in terms of text to maths conversion and then ordered.

```
forEachArg(order){
    statementList,
}
```

- *Word iterator*, it is to repeat a set of statementLists a number of times. Two arguments are passed to this statement, the string and the order. The string provides the characters to look at and also sets the number of iterations unless the inner templates modify the control statements. Order decides whether the template will start at the beginning or end of the string and then allow it to work away in the reverse direction.

```
forEachChar(string,order){
    statementList,
}
```

4.2.4 Template statements

During the analysis stage, the maintainer scripts were correlated against each other and several recurring elements were identified. For each of the patterns that were identified a corresponding DSL statement is provided. In this section we present the DSL statements grouped together by their functionality.

Alternatives

Alternatives is a location of symbolic links and helper-system where associations can be named and maintained in a consistent manner. It allows distribution and package owners to group and categorise packages that provide similar functionality. For example, instead of having to refer to every email client that exists in the repositories it is possible for package maintainers to refer to the group association “email” and if it is required by the package the operating system and user can select which email client they would like to use. Similarly if a new package provides the same features as expected in the “email” category it can use the alternatives system to suggest that it is capabale of performing the actions required.

- **add_alternative(name, location)**, this is almost identical to the preceding listing and notifies the alternatives system when a file that provides a generic feature is installed or removed. The shell or the administrator can then decide which specific file of the alternatives to use, if any. A typical implementation of this statement is as follows:

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

If the file `/usr/sbin/alternatives` is executable then run it and pass the switch to install to the sym-link of the master-link `%bindir/%name1`, the name of the master-link `%name1` and the symlink to the file to be associated `%bindir/%name2`.

- **rm_alternative(name, location)**, it is used to remove alternatives in the system configuration. In particular, it removes the sym-link `name` to `location` which is a name in the alternatives directory which in turn is a link to the actual file. A typical implementation of this statement in shell script is the following:

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

If the preceding regular expression result is equal to the string “0” and if the file `/usr/sbin/alternatives` is executable ie. update-alternatives for Mandriva is installed for CM/MD based systems and alternatives for Deb based systems then run it and pass the switch to remove the name from `/etc/alternatives` from the associated path links. If it is the last type of association then it will delete the association completely. Alternatives and update-alternatives maintain a list of master and slave symbolic links.

- **update_alternative(name, location)**, this command is provided in order to change the current executable that we want to assign to a given alternative. For instance, we have two alternative Java virtual machines in our configuration and we want to change the once that has to be executed when we execute the `java` command.

Desktop

Packages that use a GUI tend to add a .desktop entry to /usr/share/applications/ containing MIME filetype information. These configurations help suggest what file association should be made with the .desktop file. File opening scripts such as xdg-open make use of the mimeinfo.cache generated from the set of .desktop files. To rebuild, add or remove associations a desktop-cache updater needs to be run after installation and prior to removal of a package. How this is achieved tends to be distribution and windowing system and desktop environment dependent.

- **update_desktopdb(location)**, it is used to update the desktop database typically after a package installation or removal. A sample implementation of the statement in Fedora is as follows:

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

Essentially, the template above checks for the existence of update-desktop-database. The Fedora template will do nothing (|| :) if `update-desktop-database`, in silent mode (`&> /dev/null`), fails.

Doc-base

Doc-base was conceived by Debian developers as a way of producing a standardised documentation system that would work with the two main helper utilities at the time *dwww* and *dhelp* and adhere to the Debian policy which was open to interpretation on this subject [4].

- **inst_doc(package, doc_file, target_doc_file_location)**, it is used to install the documentation file of a given package. Such file is denoted by the parameter `doc_file`. The implementation in Debian of such statement is as follows:

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

[4] `http://www.fifi.org/doc/doc-base/doc-base.html`

Debian checks for the existence of install-docs in any of the `$PATH` directories and that the state is "configure". If that is the case it will run install-docs and install the file denoted by `#DOC-ID#`. This method has largely been superseeded in dh_installdocs to use alternatives. Doc-base file processing is now handled by file-triggers. Debhelper > 7.2.3 will remove these code snippets. Installs documentation for the package into /usr/share/doc/<packagename>.

- **rm_doc(doc_file, doc_file_location)**, it is used to remove the documentation file of a given package. Such file is denoted by the parameter `doc_file` and it is located in `doc_file_location`. The implementation in Debian of such statement is as follows:

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

The code above checks to see if the state is either "remove" or "upgrade". If this is the case and install-docs exists in any of the `$PATH` directories run it and remove the documentation denoted by `#DOC-ID#`. Similar to the previous listing except this deregisters the documentation and removes the doc files.

Emacs

Emacs templates allow the installation of scripts without the necessity for an administrators account because they call a package installer provided by Emacs itself.

- **inst_emacs_package(package_name)**, this statement is used to install an Emacs package. In order to do this in Debian the following code has to be provided

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

This template checks to see if the state is "configure" and whether the file `emacs-package-install` under directory `/usr/lib/emacsen-common/` is executable. If this is the case then run the emacs-package-install executable and install #PACKAGE#. Emacs uses its own package installer to install LISP scripts to customise the look, layout and functionality of emacs without requiring administrator privileges. Elisp package manager does not provide dependency mappings or requirements and so may require additional correction from errors provided within emacs.

- **rm_emacs_package(package_name)**, once the file of an emacs package have been remove from the system, the configuration model has to be updated by means of this commands which remove the corresponding `EmacsPackage` instance.

GConf

GConf is a system that was created to help the management of user preferences associated with applications [5]. It stores user preferences in a configuration database that is represented in the file-system as `/etc/gconf/<version>/path`. These preferences can then be referenced to by any application using the associated key. In this way, multiple applications can rely on the preferences set by one tool. This has obvious benefits for system wide preferences but also means that the administrator can backup and maintain preferences without having to examine each program to see where the preferences are stored. Schemas collate all this configuration information as meta-data into a .schema file. They can be bundled with packages to allow default user preferences to be selected. They need to be placed in the gconf directory and then cache updater has to be called using `SIGHUP` on the GConfD daemon.

[5] `http://www.caixamagica.pt/pag/a_index.php`

- **preinst_gconf**, this command capture the recurrent code which is provided before the GConf installation. In the following the corresponding Fedora implementation is reported

```
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
```

  In particular, the code above sets up the GCONF_CONFIG_SOURCE variable. The variable requests the default path to install schemas and also sets up the deinstallation rule that should be called on removal of the package. It then uses a macro to generate a schema. It does all of this or if any stage fails then it will do nothing and likely succeed. Gconftool is used to install schemas and register them with gconf. The standard location for gconf schemas to be installed is `/etc/gconf/schemas`. Gconf-editor is used to validate the resgistration to `/usr/share/gconf/schemas`.

- **postinst_gconf**, it is used after the installation of Gconf and some corresponding implementations are reported. For instance the Debian specification is as follows:

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

  The template above checks to see if the installer state is "configure" and if so then after the install of the files it registers the schema, `#SCHEMAS#`, for GConf preferences.

  The Fedora implementation of the same statement is reported in the following. It sets up the `GCONF_CONFIG_SOURCE` variable and then runs the macro `%_sysconfdir/gconf/schemas/[NAME].schemas` and pipes the output into nothing or does nothing.

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

- **postrm_gconf**, it is provided to completely remove of gconf settings. Its Debian implementation is as follows

```
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
```

  The template checks for the existence of a directory of name `/etc/gconf/schemas/` and deletes the schemas as defined by `#SCHEMAS#` by calling `rm -f` on them. It then deletes the directory `/etc/gconf/schemas/` and its immediate parent `/etc/gconf/`. The Fedora implementation of the same statement is given in the following

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

  It gets the GConf default directory and then creates an uninstall rule for the particular schema.

- **prerm_gconf**, it is provided to remove all the gconf schemas in the system. A sample Debian implementation of this statement is as follows

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

The template looks to see if the state is either “remove” or “upgrade” and if so before it removes the files it requests the removal of all schemas defined by `#SCHEMAS#` using `gconf-schemas --unregister #SCHEMAS#`.

Icons

An icon theme directory is a location where Gnome can find an index.theme file. Using this the cache-manager can traverse the directory in a structured manner to find icon files. Icons are normally moved to `/usr/share/icons/hicolor` and then an memory map cache of the files is created [6]. This speeds up the look up time for icons commonly used in Gnome and certain other applications that utilise the cache [7].

- **update_icons**, this command is introduced to update the cache of the icons available in the system. The Fedora implementation of this command is as follows

```
%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
```

The previous Fedora template modifies the `%_datadir/icons/hicolor` file to today’s time and date if it exists. Then it checks for an executable file `%_bindir/gtk-update-icon-cache` and if it is run it silently (–quiet) for the file that has just been modified. If that file didn’t exist it will do nothing instead.

Info

Info files are Texinfo files formatted to work with the `info documentation` program [8]. They use a combination of ASCII text and @-commands to instruct the reader program to format the text in a particular way. When the command is run any menu entries in the Info files are merged into the top-level Info file [9].

- **inst_info(info_file_location)**, installs a given info file in to a predefined directory of the system. In Fedora, this command is implemented as follows:

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

In particular, it runs the `/sbin/install-info` command with the parameters defined in the spec file as %_infodir/%name.info and %_infodir/dir. It installs the info file as defined into a directory of the same name as the source but under the folder dir.

- **rm-info(info_file_location)**, it removes the given info file from the predefined directory of the system. In Fedora, this command is implemented as follows

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

In particular, Fedora checks to see if the regular expression matches 0 and if so to run `/sbin/install-info` and delete the info file from under the `%_infodir/dir` directory, or if that fails do nothing.

[6] `http://library.gnome.org/devel/gtk/unstable/gtk-update-icon-cache.html`

[7] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=369755`

[8] `http://gd.tuwien.ac.at/.vhost/xemacs.org/Documentation/21.5/html/texinfo_2.html#SEC2`

[9] `http://www.gnu.org/software/hello/manual/texinfo/Invoking-install_002dinfo.html`

Init

Linux services can be started, stopped and reloaded with the use of scripts typically stocked in `/etc/init.d/`. When installing a new service under debian, the default is to enable it. So for instance, if you just installed `apache2` package, after you installed it, `apache` service will be started and so will it be upon the next reboots. If you do not use apache all the time, you might want to disable this service from starting up upon boot up and simply start it manually when you actually need it by running for instance the command `/etc/init.d/apache2 start`. You could either disable this service on boot up by removing any symbolic links in `/etc/rcX.d/SYYapache2` or by using update-rc.d which will take care of removing/adding any required links to /etc/init.d automatically. Hence if you want to totally disable `apache2` service by hand, you would need to delete every single link in `/etc/rcX.d/`. Using `update-rc.d` it is as simple as `update-rc.d -f apache2 remove`. In the following a number of command are proposed to manage the `init` configuration of the considered system.

◇ **postinst_init(service_name)**, this command is used to update the initialization configuration and start a service which have been previously installed. In Debian, to perform this task the following code is provided

```
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
```

In particular, this template checks to see if an excutable script file exists in `/etc/init.d/` by the name of `#SCRIPT#` and if so call `update-rc.d` on the script of the same name with the initialisation parameters `#INITPARAMS#` and output all standard info to null. If `invoke-rc.d` is executable it then runs the script by using the parameter start and if that fails it passes it to an error handler defined in the script as `#ERRORHANDLER#`. Otherwise it will call `/etc/init.d` to start the script using the start parameter and pass it to the same error-handler.

◇ **postinst_init_nostart(service_name)**, this command is provided in case the maintainer prefers to update the initialization configuration without restarting all the default services. In Debian, the implementation of this command is as follows

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

Essentially, the Debian script checks for an executable file called `#SCRIPT#` under `/etc/init.d/` and if it is found the script calls `update-rc.d` on the script and passes it initialisation parameters `#INITPARAMS#` and if there are any errors it also provides an error handling mechanism, namely `#ERROR_HANDLER#`.

◇ **postinst_init_restart(service_name)**, this command is provided in case the maintainer prefers to update the initialization configuration and restart all the default services. In Debian, the implementation of this command is as follows

```
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
```

The script first checks for an executable script #SCRIPT# under /etc/init.d and if so calls update-rc.d on the script passing the initialisation parameters #INITPARAMS#. It then checks to see if a non-empty string has been passed as the second command line parameter and if it is it creates an external variable _dh_action=restart otherwise it sets the external variable to _dh_action=start. The next part of the script checks to see if invoke-rc.d exists in the $PATH directories and if it is executable. If it is then it is called with #SCRIPT# and performs the action as stated by the variable chosen before $_dh_action. If invoke-rc.d does not exist in the path and is not executable /etc/init.d/#SCRIPT# is called instead with the same parameter as chosen before. If either of these methods fail then an error handler is provided to deal with the failure #ERROR_HANDLER#. The script basically tries to invoke a script with either a start or restart method after installing it and if it can't use an invoker tries to run the script directly.

- **postrm_init(service_name)**, this command deletes the init scripts of a given service which has been deleted from the system. The Debian implementation of this command is as follows

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

This template looks to the see if the state is equal to "purge" and if so it calles update-rc.d on the script #SCRIPT# with the action to remove it, namely remove.

- **prerm_init(service_name)**, prior the removal of a given service, this command has to be executed to stop the execution of such a service in the system. The Debian implementation of this command is the following

```
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
```

This template checks for an executable script at /etc/init.d/#SCRIPT# and also checks the state for "remove". If both conditions are met then an executable invoke-rc.d is searched for in $PATH. If it does exist then it is run with the parameter stop or otherwise /etc/init.d/#SCRIPT# is run directly with the command stop. For both methods an error handler is provided in the form of #ERROR_HANDLER#

Install shared libraries

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. Every shared library has a special name called the "soname". The soname has the prefix "lib", the name of the library, the phrase ".so", followed by a period and a version number that is incremented whenever the interface changes. Every shared library also has a "real name", which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier. The key to managing shared libraries is the separation of these names. Programs, when they internally list the shared libraries they need, should only list the soname they need. Conversely, when you create a shared library, you only create the library with a specific filename (with more detailed version information). When you install a new version of a library, you install it in one of a few special directories and then run the program ldconfig.

ldconfig examines the existing files and creates the sonames as symbolic links to the real names, as well as setting up the cache file `/etc/ld.so.cache`.

- **update_libs()**, it is provided to update after library installations or removals the cache file containing the references to the shared libraries which are installed in the system. A sample Debian implementation of this command is as follows

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

This Debian template runs `ldconfig` which configures the dynamic linker run time bindings. It creates the necessary links and cache to the most recent shared libs found in the dirs specified to it and in this case as no directories specified, all standard directories: `/etc/ld.so.conf`, `/lib`, and `/usr/lib`.

Menu

Once a package has been installed, or removed eventually it is necessary the updating of the window manager configuration files to make the new program show up on. Debian exploits the `update-menus` command to automatically generates menus of installed programs for window managers and other menu programs. The following commands are provided for managing menus.

- **update_menu**, is the command which is devoted to the updating of the menu entry files. The Debian implementation of this command is as follows

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

Essentially it checks state for “configure” and also whether an executable file `update-menus` exists in the `$PATH` and if so runs update-menus.

Mime

MIME (Multipurpose Internet Mail Extensions, RFC 1521) is a mechanism for encoding files and datastreams and providing meta-information about them, in particular their type (e.g. audio or video) and format (e.g. PNG, HTML, MP3). Registration of MIME type handlers allows programs like mail user agents and web browsers to invoke these handlers to view, edit or display MIME types they don’t support directly. Fedora provides the maintainers with the `update-mime-database` program which allows packages to register programs that can show, compose, edit or print MIME types.

- **update_mime(directory)**, this command is provided to update the databases of the available mime types handler installed in the given directory of the system. The Fedora implementation of this command is the following

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

Fedora runs update-mime-database without any checks but with the parameter %_datadir/mime. If the process signals a failure then instead the script does nothing.

Modules

Linux kernel modules can provide services (called “symbols”) for other modules to use. If a second module uses this symbol, that second module clearly depends on the first module. These dependencies can get quite complex. The `depmod` tool is used and it creates a list of module dependencies, by reading each module under `/lib/modules/version` and determining what symbols it exports, and what symbols it needs. By default this list is written to `modules.dep` in the same directory. If filenames are given on the command line, only those modules are examined (which is rarely useful, unless all modules are listed).

- **update_modules(version)**, this command is a reincarnation of `depmod` since it aims at handling dependencies among loadable kernel modules. This command has to be executed after an installation or removal of kernel modules. A Debian implementation of the command is as follows

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

The template checks for the existence of a file `/boot/System.map-#KVERS#` and if so runs `depmod` with parameters “a”, “F” and the link to the `System.map-#KVERS#` reference for the kernel version that the module depends on. Otherwise the script returns true.

Scrollkeeper

ScrollKeeper is a cataloging system for documentation on open systems. It manages documentation metadata (as specified by the Open Source Metadata Framework (OMF))[10]) and provides a simple API to allow help browsers to find, sort, and search the document catalog. It will also be able to communicate with catalog servers on the Net to search for documents which are not on the local system[11].

- **update_scrollkeeper**, once new documentation files have be copied in the OMF directory, the scrollkeeper database have to be updated. This command is provided for this purposes and it searches the scrollkeeper OMF directory to identify if any files were added, removed, or modified and updates its internal database files to reflect any changes. This command exploits the `scrollkeeper-update` command as for instance in the following Debian scripts

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

It checks to see if the first argument passed to it is equal to the string “configure” and check whether `scroll-keeper` is in the `$PATH$` directories. If this is the case it runs `scrollkeeper-update` in quiet mode, “-q”.

SGML catalog

The Standard Generalized Markup Language (SGML) is an ISO-standard technology for defining generalized markup languages for documents [12]. Almost any FOSS system support SGML catalogs which can be installed and removed like any other software package. In this respect, a number of commands have been introduced to support SGML catalog installations and removals. Catalogs in this context are logical structures that contain links between Uniform Resource Identifiers (URIs) and external SGML identifiers. Catalogs in this context exist in other markup languages including XML [13]. They provide a 2-way mapping of key-value pairs. Catalogs in this way can utilise resources that may exist on an arbitrary system either locally or externally. Catalogs may physically exist across multiple catalog entry files that contain a set of catalog link entries.

- **postinst_sgmlcatalog**, this command updates the SGML catalog repository once a new catalog entry has been installed. For instance, in Debian this operation is implemented as in the following scripts

```
if [ "$1" = "configure" ]; then
    rm -f #CENTRALCAT#
```

[10] `http://metalab.unc.edu/osrt/omf/`

[11] `http://scrollkeeper.sourceforge.net/`

[12] http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language

[13] `http://www.oasis-open.org/committees/entity/spec.html#s.terminology`

```
    for ordcat in #ORDCATS#; do
        update-catalog --quiet --add #CENTRALCAT# ${ordcat}
    done
    update-catalog --quiet --add --super #CENTRALCAT#
fi
```

This scripts checks to see if the state is “configure” and if so it force removes `#CENTRALCAT#`. The #CENTRALCAT# is the location where the catalog entry files for packages and applications that wish to use the catalog entries, reside. It is therefore specific to a package. It then looks in every `#ORDCATS#` (an ordered list of all the catalog files that are monitored on the system [14]) directory and calls `update-catalog` in quiet mode to add `#CENTRALCAT#` to the folders specified by `$ordcat`. Once done it calls `update-catalog` to add `#CENTRALCAT#` entries into the SGML super catalog `/etc/sgml/catalog` which is akin to an index of the catalogs.

- **postrm_sgmlcatalog**, this removes the catalog and the backup of the catalog for the package. Any references to any variables in either of these catalogs will no longer be available.

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

If the first argument passed to it is equal to “purge” then it forces the removal of `#CENTRALCAT#` and `#CENTRALCAT#.old` files.

- **prerm_sgmlcatalog**, on the removal or upgrade of a package that has an associated catalog it is imperative that the super catalog database is refreshed so that the old catalogs are not refered to. The entries for the package catalog file are thus removed. If the files are to be removed there will no longer be an entry in the super catalog and if the script is an upgrade the reference to the new catalog would have been created before so the old one no longer points to a valid catalog.

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

This Debian template calls update-catalog to remove the file #CENTRALCAT# from the super SGML catalog located at /etc/sgml/catalog.

udev

udev is a generic kernel device manager. It runs as a daemon on a Linux system and listens to uevents the kernel sends out (via netlink socket) if a new device is initialized or a device is removed from the system. The system provides a set of rules that match against exported values of the event and properties of the discovered device. A matching rule will possibly name and create a device node and run configured programs to set-up and configure the device.

For the following examples #OLD# refers to the old package configuration rule which may or may not have been installed on the system. #RULE# refers to a symbolic link in the `/etc/udev/rules.d` directory that links to a file with configuration settings such as what to name a device and which additional commands to run [15].

- **preinst_udev**, when installing or upgrading a package through a call from udev it is important to check if the old package configuration exists and whether or not it is for the same version as the package. If it is the same as the package to be installed, to avoid a file conflict the existing file is removed. If there are any rules that exist for the device in `/etc/udev/rules.d` then they are similarly overwritten.

```
if [ "$1" = install ] || [ "$1" = upgrade]; then
    if [ -e "#OLD#" ]; then
```

[14] `http://www.mail-archive.com/debian-sgml@lists.debian.org/msg00906.html`

[15] `http://reactivated.net/writing_udev_rules.html`

```
        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
```

If the first parameter passed to the Debian template is either “install” or “upgrade” then two conditional loops take place. The first checks for the existence of a file #OLD# and looks for the equivalence of the old files md5sum with that of the latest package as provided by dpkg-query. If so it removes the #OLD# file. The second loop checks that the file #RULE# exists and is a symbolic link and if so removes it.

◇ **postinst_udev**, this command checks if an old configuration exists and if so checks to see if a backup has been made and if it does moves it to *.dpkg-new. The old configuration is then preserved to #RULE#.

```
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
```

If the first argument passed into this template is “configure” and there exists a file #OLD# then the following commands are run. It firstly prompts the user as to what is happening, “Preserving user changes to #RULE#...”. It then checks for the existence of the file #RULE# and if so moves that file to #RULE#.dpkg-new. The old file, #OLD#, is then moved to #RULE#.

usrlocal

In general, the `/usr/local` hierarchy is for use by the system administrator when installing software locally. It needs to be safe from being overwritten when the system software is updated. It may be used for programs and data that are shareable amongst a group of hosts, but not found in `/usr`. In fact, locally installed software must be placed within `/usr/local` rather than `/usr` unless it is being installed to replace or upgrade software in `/usr`.

◇ **postinst_usrlocal**, it looks through a file and at each line checks to see if a directory as defined in that line exists. If not it creates a directory and changes the attributes of the directory to match that found in the file listing. If other data is found then the script stops. This command is seldom run.

```
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
```

The Debian template reads all the lines and prserves whitespace,"set – $line". The script then uses the 4 parameters to define the directory, mode, user and the user's group. If the directory `$dir` does not exist then it is made and the user and group are set up for the folder and the permissions as well.

- **prerm_usrlocal** This template is used to remove empty directories that are part of the the filesystem that the maintainer believes should be removed.

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

This template loops through a list of directories, dir, and removes the directories until the delimiter DATA is found.

Users and Groups

Linux groups are a mechanism to manage a collection of computer system users. All Linux users have a user ID and a group ID and a unique numerical identification number called a userid (UID) and a groupid (GID) respectively. Groups can be assigned to logically tie users together for a common security, privilege and access purpose. It is the foundation of Linux security and access. Files and devices may be granted access based on a users ID or group ID. In the following a number of commands are provided for managing users and groups

- **user_add(user_name, group_name, homedir)**, this command is used to add the given `user` to the given `group`. If the user already exists in the system, it is simply added to the considered group. The parameter `homedir` specifies the home directory of the added user. A sample implementation of this script in Fedora is as follows

```
%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
```

This template gets the current users `GROUPNAME` or gets the default username for a newly added group `GROUPNAME` is it doesn't exist. The user is then added to group `GROUPNAME` and the main directory for that user is set as `HOMEDIR`. A comment to help describe the new user is also added as well as the desired `USERNAME`.

- **user_remove(user_name,group_name)**, this command is used to remove a given user from an existing group, or definitely remove it from the system. In particular, if the `group` parameter is missing, the specified `user` will be removed from the system, otherwise `user` will be removed from `group`. A sample implementation of user removal in Fedora is as follows

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

If the first parameter is equal to 0 then `/usr/sbin/userdel` is used to delete the user `USERNAME`.

- **group_add(group_name)**, this command is like the shell command `groupadd` used to adds a group in the system.

- **group_remove(group_name)**, this command is used to remove the specified group in the system. It is like the shell command `groupdel` and a sample use of it in Fedora is as follows

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

This template checks to see if paramater passed to it is equal to 0 and if so `/usr/sbin/groupdel` is called to delete `USERNAME` from the group listings.

Windows manager

The X Window System, known simply as “X”, is a portable, network-transparent window system which runs on many different computers. There have been numerous versions of the X Window System, but it was not until the eleventh version, known simply as “X11”, that it was widely released and began to gain the popularity it enjoys today. Since then there have been many further releases which added extra functionality while attempting to remain largely backwards compatible. The current release is the sixth one, and is known as “X11R6” or simply as “R6”. One of the guiding philosophies of The X Window System (and also UNIX itself) is that its functionality is achieved through the co-operation of separate components, rather than everything being entwined in one huge mass. The advantage of this is that a particular part of the system can be changed simply by replacing the relevant component. The best example of this is the concept of a window manager which is essentially the component which controls the appearance of windows and provides the means by which the user can interact with them. Virtually everything which appears on the screen in X is in a window, and a window manager quite simply manages them. A large number of window managers have been developed, which between them provide a large range of different appearances and different behaviours. Furthermore, most of these window managers are themselves heavily customisable. This means that a newcomer to X has firstly a choice of window manager, and then a choice of the precise configuration of the chosen window manager.

In the following a number of commands are proposed to manage the installations and removals of windows managers.

◇ **postinst_wm(wm_location,manual_location)**, once a given window manager has been installed in the system, this command has to be used to update the alternatives 4.3.2 related to the `x-window-manager` command. A sample implementation of this command in Debian is as follows

```
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
```

If the parameter passed to the template is “configure” then update-alternatives is called to install the x-window-manager `#WM#` to `/usr/bin/x-window-manager` and extracting the manual `#WMMAN#` to `/usr/share/man/man1/x-window-manager.1.gz`.

◇ **postinst_wm_noman(wm_location**, if the maintainer does not want to provide the manual of the installed window manager, this command can be used. A sample implementation in Debian is the following

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

If the parameter passed to this template equals “configure” then `update-alternatives` is called to install an `x-window-manager` named `#WM#` to the window manager `/usr/bin/x-window-manager` with priority `#PRIORITY#`.

◇ **prerm_wm(wm_location)**, in order to remove a window manager, it has to be firstly deleted in the alternatives. This command is provided for this purpose as also shown in the following Debian implementation

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

The template checks to see if the first parameter sent to it is equal to “remove”. If so update-alternatives is run to remove the x-window-manager named #WM#.

Xfonts

X includes two font systems: the original core X11 fonts system, which is present in all implementations of X11, and the Xft fonts system, which may not be distributed with implementations of X11 that are not based on X11R6.8.2

The core X11 fonts system is directly derived from the fonts system included with X11R1 in 1987, which could only use monochrome bitmap fonts. Over the years, it has been more or less happily coerced into dealing with scalable fonts and rotated glyphs. Xft was designed from the start to provide good support for scalable fonts, and do so efficiently. Unlike the core fonts system, it supports features such as anti-aliasing and sub-pixel rasterisation. Perhaps more importantly, it gives applications full control over the way glyphs are rendered, making fine typesetting and WYSIWIG display possible. Finally, it allows applications to use fonts that are not installed system-wide for displaying documents with embedded fonts.[16].

Fonts are located in a set of well-known directories that include all of X11R6.8.2’s standard font directories (`/usr/X11R6/lib/X11/lib/fonts/*`) by default) as well as a directory called `.fonts/` in the user’s home directory. Installing a font is as simple as copying a font file into one of these directories. However, there is a cache to be maintained always updated which refers to the installed fonts.

To support the installation and the removal of fonts the following command is provided

- **update_xfonts**, once a new font has been installed or removed from the system, this command has to be executed in order to updated the cache which maintain the status of the directory containing the installed fonts. A sample implementation in Debian of this command is as follows

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

The template checks for a file in the `$PATH` named `update-fonts-dir` and if found it runs the commands set by `#CMDS#`. `#CMDS#` are a list of commands that the package maintainer would like to run after the font cache has been updated. It is a mechanism by which the package manager can use user preferences to keep their configurations. The risk with updating the fonts is that by updating the values incorrectly it could lead to a non-functioning GUI from which the user would have to recover.

4.2.5 Tagging statements

As said at the beginning of this chapter, the DSL we are proposing derives from a deep analysis of many Linux distributions. This analysis aimed at finding in maintainer scripts the parts which are automatically generated and discovering common recurrences. For instance, concerning Debian Lenny we analyzed all the 25’440 available maintainer scripts. We discovered that about 2/3 of them are composed only of lines generated using autoscript mechanisms. In particular, 16’348 (64,3%) scripts are generated, and 9’061 (35,6%) are written by hand. We have further investigated the hand written scripts in order to find additional commonalities that can give place to further commands in the DSL. We came up with additional templates by covering in total the 66% of the existing 25’440 scripts. Concerning

[16] `http://www.x.org/X11R6.8.2/doc/fonts.html`

RPM based distributions, we analyzed Fedora and we considered all the available 2'038 maintainer scripts. Our analysis has shows that 1'962 (93,6%) scripts are automatically generated starting from recurring templates.

In this respect, according to the analysis just summarized the DSL commands we provided until now is not able yet to specify 100% of the maintainer scripts. There are scripts which contains specific commands with unique occurrences which are not covered by the DSL. However, to enable the simulation of those scripts which contain also such kind of commands, we provide a tagging mechanism which can be used by maintainers to describe the behavior of the commands which are not in the DSL. In particular, maintainers have the possibility to specify how the configuration model changes once a given command (not yet captured by the DSL) is executed.

The tagging commands which are provided for this purpose permit the specification of additions, deletions of the configuration models and are based on the following syntax

```
-- Addition and deletion of metaclass instances
add<metaclassName>(<metaclass feature values>)
delete<metaclassName>(<metaclass feature values>)

-- Addition and deletion of attribute values
add<metaclassName>_<attribute>(<element>,<attribute value>)
delete<metaclassName>_<attribute>(<element>)

-- Addition and deletion of references
add<metaclassName>_<reference>(<source element>,<target element>)
delete<metaclassName>_<reference>(<source element>,<target element>)
```

In other words, maintainer has the means to specify the differences [CDP07] between two configuration models which occur because of the execution of the considered scripts. For instance, if maintainers want to specify the following code

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

the DSL does not have a template statement to directly specify it. In fact, the analysis we performed did not detect enough occurrences of the snippet of code above to justify the need for a specific metaclass. However, maintainers can specify its behavior as follows

```
1 #<%
2 addPackageSetting_dependences(apache2,php5);
3 addEnvironment_runningServices(env,apache2);
4 #%if [ -e /etc/apache2/apache2.conf ] ; then
5 #%    a2enmod php5 >/dev/null || true
6 #%    reload_apache
7 #%fi
8 #%>
```

In this case, the maintainer specifies the behaviour of the script code by enclosing it in a `#<%` ... `#%>` block. Immediately after its open tag, a sequence of tag commands are given in order to specify how the configuration model changes if the considered script code is executed. In this example, the execution of `'a2enmod php5 >/dev/null || true'` modifies the configuration model by adding a new dependency between the package settings of the `apache2` and `php5` packages (see line 2 above). Then `apache2` is reloaded and this implies the addition of `apache2` as a running service in the environment (see line 3 above). In other words, maintainers has the possibility to specify the semantics of script codes (which no templates are available for) in terms of changes which occur in the configuration model.

4.3 MANCOOSI DSL: Semantics

In this section we provide the semantics of the MANCOOSI DSL. The semantics of a DSL captures the effect of "sentences" of the language. As previously said, here we are interested in *dynamic semantics* which deals with the behavior expressed by a language term (what something *does*), contrarily to the *static semantics* which express the structural meaning of a language term (what something *is*). Specifying the semantics of languages is a difficult task and there is not a generally accepted formalism for it. Over the last decades several semantics formalisms have been proposed but none emerged as universal and commonplace, as for instance happened to the EBNF for context-free syntaxes. Depending on the application purpose (formalization, simulation, verification, consistency checking, etc.) a number of formalisms are available (Object-Z [Smi00], ASMs [BÖ2], Structured Operational Semantics (SOS) [Plo81], etc.). In general, there are at least four ways in which we can describe the semantics of a software language (see [ZX04] for a survey of semantic description frameworks) [Kle07]:

- *Denotational*, that is by constructing mathematical objects (called denotations or meanings) which represent the meaning of the program/model;
- *Operational*, that is by describing how a valid program is interpreted as sequences of computational steps. The sequence of computational steps is often given in the form of a state transition system, which shows how the runtime system progresses from state to state;
- *Translational*, that is by translating the program into another language that is well understood;
- *Pragmatic*, that is by providing a tool that executes the program/model. This tool is often called a reference implementation.

Concerning the MANCOOSI DSL, the semantics is specified in an operational way in terms of model-to-model transformations. In particular, for each command introduced in the previous section, a corresponding model transformations is given in order to describe exactly the behavior of each statement of the language when executed or simulated. Hence, the execution of the scripts involved into an upgrade implies the application of model transformations on the source configuration model.

The remaining of the section is organized as follows: next section provides the reader with preliminary concepts related to model transformations and ATL which is the language which has been used for implementing transformations. Section 4.3.2 uses such preliminary concepts and defines the semantics of all the DSL commands presented in the previous section.

4.3.1 Model transformations and ATL in a nutshell

The MDA guide [OMG03] 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 Figure 4.3, a model transformation programs take as input a model conforming to a given source meta-model and produces as output another model 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.

Many languages and tools have been proposed to specify and execute transformation programs. In 2002 OMG issued the Query/View/Transformation request for proposal [OMG02] 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 have been proposed both from academia and industry. The paradigms, constructs, modeling approaches, tool support distinguish the proposals each of them with a certain suitability for a certain set of problems.

In this document ATL (ATLAS Transformation Language) [JK05] will be considered. It is a hybrid model transformation language containing a mixture of declarative and imperative constructs. The

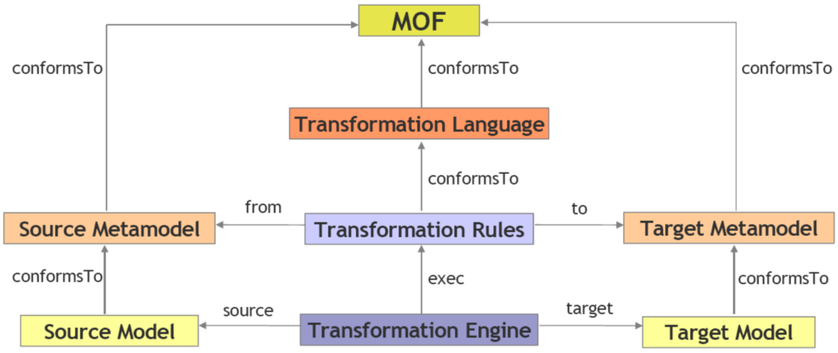


Figure 4.3: Basic Concepts of Model Transformation

former allows to deal with simple model transformations, while the imperative part helps in coping with transformation of higher complexity. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. During the execution of a transformation source models may be navigated but changes are not allowed. Target models cannot be navigated.

Transformation definitions in ATL form *modules*. A module contains a mandatory *header* section, *import* section, and a number of *helpers* and *transformation rules*. Header section gives the name of a transformation module and declares the source and target models (lines 1-2, Figure 4.4). The source and target models are typed by their meta-models. The keyword `create` indicates the target model, whereas the keyword `from` indicates the source model. In the example of Figure 4.4 the target model bound to the variable `OUT` is created from the source model `IN`. The source and target meta-models, to which the source and target model conform, are `PetriNet` and `PNML` [BCvH+03] respectively.

Helpers and transformation rules are the constructs used to specify the transformation functionality. Declarative ATL rules are called *matched rules*. They specify relations between *source patterns* and

```
 1 module PetriNet2PNML; create OUT : PNML from IN : PetriNet;
 2 ...
 3 rule Place {
 4         from
 5                 e : PetriNet!Place
 6                 --(guard)
 7         to
 8                 n : PNML!Place
 9                 (
10                         name <- e.name,
11                         id <- e.name,
12                         location <- e.location
13                 ),
14                 name : PNML!Name
15                 (
16                         labels <- label
17                 ),
18                 label : PNML!Label
19                 (
20                         text <- e.name
21                 )
22 }
```

Figure 4.4: Fragment of a declarative ATL transformation

target patterns. The name of a rule is given after the keyword `rule`. The source pattern of a rule (lines `5-7`, Figure 4.4) specifies a set of *source types* and an optional *guard* given as a Boolean expression in OCL. A source pattern is evaluated to a set of matches in source models. The target pattern (lines `8-22`, Figure 4.4) is composed of a set of *elements*. Each of these elements (e.g. the one at lines `9-14`, Figure 4.4) specifies a *target type* from the target meta-model (e.g. the type `Place` from the `PNML` meta-model) 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 is used to initialize the feature. In some cases complex transformation algorithms may be required and it may be difficult to specify them in a declarative way. For this issue ATL provides two imperative constructs: *called rules*, and *action blocks*. A called rule is a rule called by other ones like a procedure. An action block is a sequence of imperative instructions that can be used in either matched or called rules. The imperative statements in ATL are the well-known constructs for specifying control flow such as conditions, loops, assignments, etc.

4.3.2 Operational Semantics using ATL

The operational semantics of a software language describes what happens in a system when a program of that language is executed. It can be done in the form of a set of rules that govern an abstract machine that is able to execute any syntactically correct sentence of the language. Execution means the processing of data. Therefore a semantics description should explain *(1)* the data being processed, *(2)* the processes handling the data, and *(3)* the relationship of these two with the possible sentences of the language [Kle07]. Typically, *(1)* and *(2)* are the runtime system, *(3)* is called the semantic mapping.

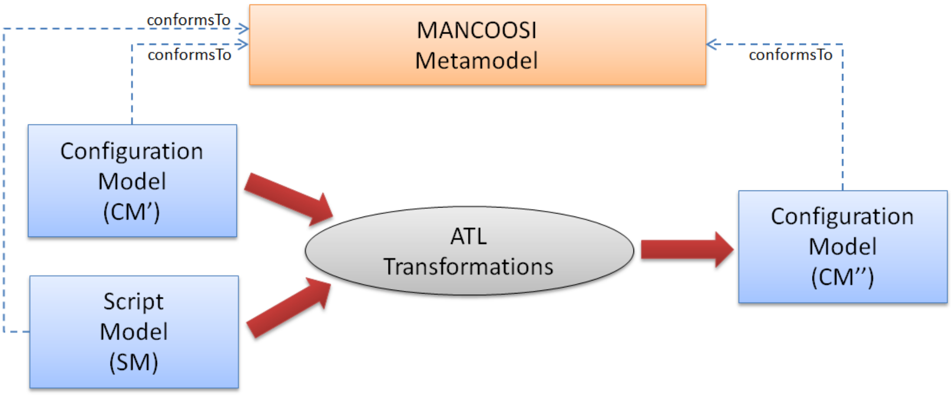


Figure 4.5: Operational Semantics using ATL

The runtime system can be described using the formalism of metamodeling. For instance, to define an Object-Oriented execution semantics a proper metamodel can be defined to describe what in compiler terminology is known as the heap, the data part of the runtime system, and the stack, the process part of the runtime system. The semantic mapping can be given in terms of model transformation rules. Each rule can be applied only when a certain syntactic construction is present in the program and when at the same time a certain runtime state has been reached. For example, the rule describing assignment is executed only when the program counter has reached an assignment statement in the (abstract syntax graph of the) program and the value of the expression in the right hand side of the assignment is available on the heap simultaneously.

In our case, the runtime system is given in terms of a configuration model which represents the current state of the system in which a given maintainer script has to be executed. Both the configuration and the script models conform to the MANCOOSI metamodel (see Figure 4.5) already presented in the Deliverable D2.1 [17]. The semantic mappings are given in terms of ATL transformations. In particular, for each command of the DSL a corresponding transformation rule is defined. In general, concerning commands which imply the addition of new configuration elements (e.g. `add_alternatives`), or the deletion of already existing ones (e.g. `rm_alternatives`), specific ATL *called rules* are defined. In fact, as previously said called rules are transformation rules which are invoked by other rules and lack source patterns. This characteristics fits well with the semantics of addition or deletion of configuration elements, since no source patterns have to be considered for their application. On the contrary, concerning the commands

[17] `http://www.mancoosi.org/deliverables/d2.1.pdf`

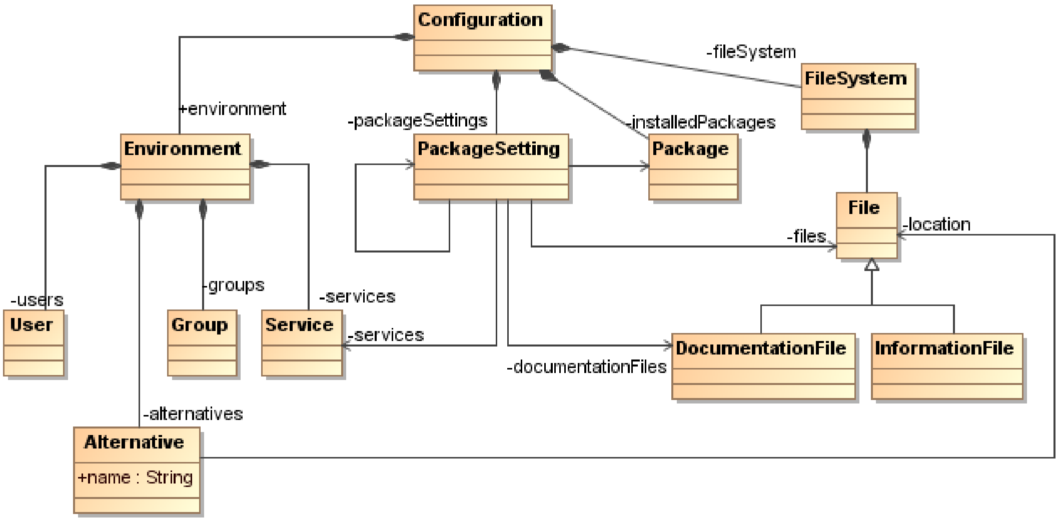


Figure 4.6: Fragment of the Configuration Metamodel

which implies the update of existing configuration elements, normal matched rules are specified since source and target patterns have to be specified. For instance, the `update_desktopdb` command considers the desktop db of the old configuration, and update it leading to the new configuration.

ATL has been adopted for specifying the semantics of the proposed DSL for many reasons, even though it is not the unique choice. It is worth mentioning at least two main motivations:

- in a Model Driven Engineering setting, ATL is the “natural” choice to specify model manipulations and transformations. More and more people is adopting this technology which is supported by a large and active community;
- we could have specified the semantics of our language by means of other technologies even more formal. However, the simulator and the failure detector (which will be provided by the end of the project and will be the focus of the deliverable D2.3) exploit ATL for implementing and executing the configuration model modifications. This means that a big part of the semantic specification that will be provided in this document will be refined and reused in the remaining of the project.

In the following the fragment of the MANCOOSI metamodel devoted to the configuration representation is summarized in order to better understand the DSL semantics specification which is provided in the rest of the section. In particular, maintainer scripts act on a configuration model which contains the elements reported in Figure 4.6. The `Environment` metaclass enables the specification of loaded modules, shared libraries, and running process as in the sample configuration reported in Figure 4.7. 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.

Note that Figure 4.6 depicts a fragment of the Configuration metamodel. The semantic specification of the DSL that will be given in the rest of the section, will exploit further metaclasses and structural features which are reported in the Appendix A.

All the services provided by a system can be used once the corresponding packages have been installed (see the association between the `Configuration` and `Package` metaclasses in Figure 4.6) and properly configured (`PackageSetting`). Moreover, the configuration of an installed package might depend on other package configurations. For example, considering the PHP5 upgrade mentioned in the previous section, the instances `ps1` and `ps2` of the `PackageSetting` metaclass in Figure 4.7 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 Figure 4.7) and both are also associated with the corresponding files which store their configurations. Note that at the level of package meta-information

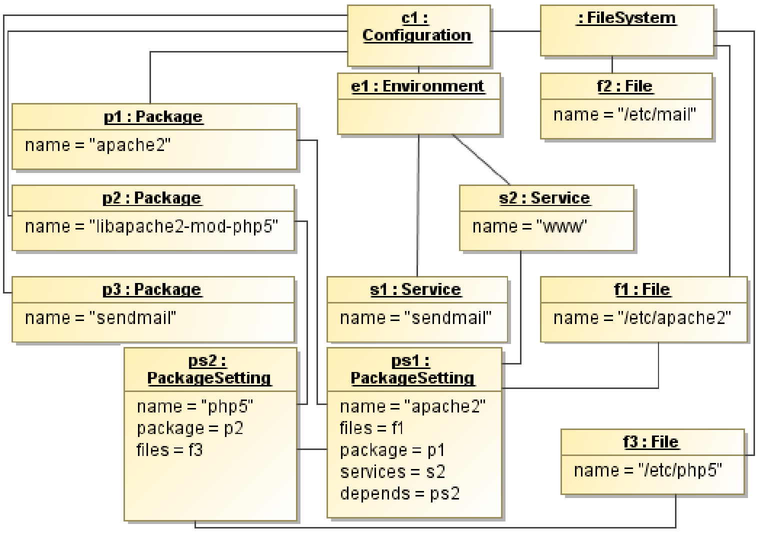


Figure 4.7: Sample Configuration model

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

The other concepts of the metamodel in Figure 4.6 will be described throughout this section during the presentation of the ATL transformations which relies on them.

The rest of the section specifies the semantics of the identified template commands and the discussion is organized with respect to the command categorization already done for presenting the concrete syntax of the DSL. For each command, ATL transformation rules are given in order to explain how the execution of the considered command changes the source configuration.

Alternatives

◇ **add_alternative(name, location)**, this command adds a new alternative in the system configuration. In particular, `name` is the name of the considered alternative (e.g. *java*) and `location` refers to the real executable in the file system (e.g. `/usr/j2se/bin/java`). If `name` already exists in the source configuration, a new reference from the existing `name` alternative to the `location` file is created (see lines 7-8), otherwise, a new alternative is also created (see lines 8 and 16-25). Two specific helpers are used, `existsAlternative`, and `getAlternative`. The former queries the source configuration and checks if exists an alternative which has the name passed as parameter (see line 7). The latter returns the alternative which has the name passed as parameter (see line 8).

```
1  rule add_alternative(name:String, location:File){
2      using {
3          newAlternative : OUTConfiguration!Alternative = OclUndefined;
4      }
5
6      do {
7          if thisModule.existsAlternative(name) {
8              thisModule.getAlternative(name).location <- location;
9          } else {
10             newAlternative <- thisModule.createAlternative(name,location);
11         }
12         t;
13     }
14 }
```

```

rule createAlternative(name:String, location:File){
    to
        t : OUTConfiguration!Alternative (
            name <-name,
            location <- location
        )
    do {
        t;
    }
}
```

- ◇ **rm_alternative(name, location)**, this commands remove an alternative in the source configuration.

```
rule rmAlternative{
    from
        s: INConfiguration!Alternative(
            s.name = name and s.location = location
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the Alternative s is not copied to the target configuration
    }
}
```

- ◇ **update_alternative(name, location)**, this command is provided in order to change the current executable that we want to assign to a given alternative. For instance, we have two alternative Java virtual machines in our configuration and we want to change that which has to be executed when we execute the `java` command.

```
rule update_alternative(name, location){
    using {
        alternative : OUTConfiguration!Alternative = OclUndefined;
    }
    do {
        alternative <- thisModule.getAlternative(name);
        alternative.current <- location;
    }
}
```

Desktop

- ◇ **update_desktopdb(location)**, this command updates the cache which maintains the references to the installed applications. If no parameters are passed default locations are considered. The desktop db is maintained by means of the `DesktopDB` elements which has the reference `applications` (see Figure 4.6) to maintain the references to the installed desktop applications. In this respect, the helper `getDesktopApplicationFiles` is used. It searches in `location` of the file system for all the `.desktop` files.

```
rule update_desktopdb(location){
    from
        s : INConfiguration!DesktopDB
    to
        t : OUTConfiguration!DesktopDB (
            applications <- thisModule.getDesktopApplicationFiles(location);
        )
}
```

Doc-base

- ◇ **inst_doc(package, doc_file_location, target_doc_file_location)**, this command installs a documentation file for the given package. The execution of this command creates a new instance of the metaclass DocumentationFile and update the documenationFiles relation in the given package to include the new file. In the following transformation, the new documentation file is created by means of the createDocumenationFile called rule (see line 8) which takes the location of the document as parameter and create a new instance of the DocumentationFile metaclass. The helper getPackage is used to query the configuration model and select the passed package whose documentationFiles relation will be updated by adding the file newDocumentationFile as specified in line 9

```
1  rule inst_doc(package, doc_file_location){
2      using {
3          newDocumentationFile : OUTConfiguration!DocumentationFile = OclUndefined;
4          package : OUTConfiguration!Package = OclUndefined;
5      }
6      do {
7          package<-thisModule.getPackage(package);
8          newDocumentationFile <- thisModule.createDocumentationFile(doc_file_location,
              ↪target_doc_file_location);
9          package.documentationFiles <- newDocumentationFile;
10     }
11 }
```

- **rm_doc(package, doc_file_location)**, this command removes from the configuration model the considered documentation file.

```
rule rm_doc(package, doc_file_location){
    from
        s: INConfiguration!DocumenationFile(
            s.location = doc_file_location
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the DocumenationFile s is not copied to the target configuration
    }
}
%
```

Emacs

- **inst_emacs_package(package_name)**, this commands install a new emacs package. In particular, it creates a new instance of the metaclass EmacsPackage which is part of the system configuration.

```
rule inst_emacs_package(package_name){
    using {
        newEmacsPackage : OUTConfiguration!EmacsPackage = OclUndefined;
    }
    do {
        newEmacsPackage <- thisModule.createEmacsPackage();
        newEmacsPackage.name <- package_name;
    }
}
```

- **rm_emacs_package(package_name)**, once the files of an emacs package have been remove from the system, the configuration model has to be updated by means of this commands which removes the corresponding EmacsPackage instance.

```
rule rm_emacs_package(package_name){
    from
        s: INConfiguration!EmacsPackage(
            s.name = package_name
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the EmacsPackage s is not copied to the target configuration
    }
}
```

GConf

- ◇ **preinst_gconf**, this command creates the `GConf` element which will be used to manage all the gconf configuration files and create the directory `/etc/gconf/schemas`. In this respect, the helper `thisModule.createGConf` is invoked and the command `addFile('/etc/gconf/schemas')` is executed.

```
rule preinst_gconf{
    using {
        newGConf : OUTConfiguration!GConf = OclUndefined;
    }
    do {
        newGConf <- thisModule.createGConf();
    }
}
```

- ◇ **postinst_gconf**, this command registers the schema files contained in the directory `/etc/gconf/schemas`. For this purpose the helper `getGConfSchema` is used.

```
rule postinst\_gconf{
    from
        s: INConfiguration!GConf
    to
        t: OUTConfiguration!GConf(
            schemas <- thisModule.getGConfSchema()
        )
}
```

- ◇ **prerm_gconf**, it removes all the schemas from the default directory. At this stage all the GConf files registered in the `GConf` element will be removed as follows

```
rule postinst\_gconf{
    from
        s: INConfiguration!GConf
    to
        t: OUTConfiguration!GConf(
            schemas <- OclUndefined
        )
}
```

- ◇ **postrm_gconf**, this command is used to remove all schema files within the default schema directory, `/etc/gconf/schemas`. The default directory is also then deleted. For this purpose, for each file `f` in `/etc/gconf/schemas`, the command `delFile(f)` is executed. Then also the command `delFile(/etc/gconf/schemas)` is invoked.

Icons

- ◇ **update_icons**, this commands updates the `mtime` of the abstract representation we provide for the icon cache. In particular, it considers the instance of the `IconCache` metaclass and updates the `mtime` attribute. This update exploits the helper `Mtime` which returns a string which encodes the time of invocation.

```
rule update\_icons{
    from
        s: INConfiguration!IconCache
    to
        t: OUTConfiguration!IconCache (
            mtime <- thisModule.getMtime()
        )
}
```

Info

- **inst_info(info_file_location)**, this command installs an information file in the system. The execution of this command creates a new instance of the metaclass `InformationFile`. In the following transformation, the new information file is created by means of the `createInformationFile` called rule (see line 6) which takes the location of the file as parameter and creates a new instance of the `InformationFile` metaclass.

```
1 rule inst_info(info_file_location){
2     using {
3         newInformationFile : OUTConfiguration!InformationFile = OclUndefined;
4     }
5     do {
6         newInformationFile <- thisModule.createInformationFile(info_file_location);
7     }
8 }
```

- **rm_info(info_file_location)**, this command removes from the configuration model the considered information file.

```
rule rm_info(package, info_file_location){
    from
        s: INConfiguration!InformationFile(
            s.location = info_file_location
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the InformationFile s is not copied to the target configuration
    }
}
```

Init

- **postinst_init(service_name)**, this command modifies the configuration model by adding the service named `service_name` in the `services` relation of the `Boot` metaclass. This metaclass is used to model the typical `/etc/init.d` location which maintains the services which has to be started when the system is booted. In the following called rule, the helper `getServiceByName` is used in order to retrieve from the source configuration the service named `service_name`

```
rule postinst_init(service_name){
    from
        s: INConfiguration!Boot
    to
        t: OUTConfiguration!Boot(
            services <- s.services->union(thisModule.getServiceByName(service_name))
        )
}
```

- **postinst_init_nostart(service_name)**, this command is used to update the `Boot` instance in order to do not start the service named `service_name` when the system is booted.

```
rule postinst_init(service_name){
    from
        s: INConfiguration!Boot
    to
        t: OUTConfiguration!Boot(
            services <- s.services->excluding(thisModule.getServiceByName(service_name
                ↪))
        )
}
```

- **postinst_init_restart(service_name)**, this command performs the same operations of the command **postinst_init** and starts all the services which are specified in the `Boot` element of the current environment. In this respect, for each service maintained in the `services` relation of `Boot` the following called rule is executed

```
rule runService(service){
   using {
        environment : OUTConfiguration!Environment = OclUndefined;
    }
    do {
        environment <- thisModule.getEnvironment();
        environment.runningServices <- service; -- service is added as
                                                -- running
    }
```

- **prerm_init(service_name)**, prior the removal of a service from the `Boot` instance, such a service has to be stopped. In this respect, the following called rule is executed. In particular, the set of running services in the system environment is modified by excluding the service named `service_name`

```
rule prerm_init(service_name){
  using {
        environment : OUTConfiguration!Environment = OclUndefined;
    }
    do {
        environment <- thisModule.getEnvironment();
        environment.runningServices <- environment.runningServices->excluding(
            ↪thisModule.getServiceByName(service_name));
    }
```

- **postrm_init(service_name)**, this command modifies the `Boot` element to exclude the service named `service_name` from the services to be executed when the system is booted as specified in the following called rule

```
rule postrm_init(service_name){
  using {
        boot : OUTConfiguration!Booot = OclUndefined;
    }
    do {
        boot <- thisModule.getBoot();
        boot.services <- boot.services->excluding(thisModule.getServiceByName(
            ↪service_name);
    }
}
```

Make shared libraries

- **update_libs**, this command updates the configuration model by searching for new shared libraries in the modeled file `/etc/ld.so.conf`, and in the trusted directories (`/lib` and `/usr/lib`). The execution of this commands considers the reference `locations` of the `Library` instance and updates its `sharedLibraries` reference to maintain all the shared libraries which are located in such locations.

```
rule update_libs {
    from
        s : INConfiguration!Library

    to
        t : OUTConfiguration!Library (
            locations <- s.locations
            sharedLibraries <- thisModule.getSharedLibraries(s.locations);
        )
}
```

Menu

- **update_menu**, this command causes the cache of menu entries to be refreshed as specified in the following

```
rule update_menu {
    from
        s : INConfiguration!Menu

    to
        t : OUTConfiguration!Menu (
            entries <- thisModule.getMenuEntries()
        )
}
```

The class `Menu` is used to maintain the menu cache which refers to all the installed entries which are represented by means of the metaclass `MenuEntry`. The helper `getMenuEntries` searches for all the menu entries installed on the system.

- **postinst_menu**, this command checks to see if a file on the filesystem has been installed of the same name as expected and if it is executable. For us we will use the file-system meta-class to analyse whether or not a file has been registered in this location. If so the update_menu script is run and is defined above.

- **postrm_menu**, this command iterates through a list of packages provided to the script and makes the files non-executable. Update_menu script is then run afterwords which is defined above. Non-executable files are not restored by the update_menu command.

Mime

- **update_mime(directory)**, this command updates the cache of the mime type handlers installed in the system. In the configuration model, the metaclasses `MimeTypeHandlerCache` and `MimeTypeHandler` are used for this purpose. In particular, the command checks the existence of mime type handlers in the `directory` passed as parameter and for each new handler found, a new `MimeTypeHandler` instance is created. Being more precise, the following called rule defines the semantic of this command

```
rule update_mime(directory) {
    from
        s : INConfiguration!File (
            s.isMimeTypeHandler() and s.isIn(directory)
        )

    to
        t : OUTConfiguration!MimeTypeHandler (
            handler <- s,
            type <- thisModule.getMimeType(s)
        )
}
```

Different helpers are used in the previous called rule. In particular, `isMimeTypeHandler` returns `true` is the considered file is a mime type handler, `false` otherwise. The helper `isIn` returns `true` if the given file is in the directory passes as parameter. Finally, given an handler `s`, the helper `getMimeType` returns the mime type managed by the handler `s` (e.g. `application/vnd.openxmlformats`, and `application/x-java-archive`).

Modules

- **update_modules(version)**, it updates the cache of the installed modules in the system for the kernel version passed as parameter. In this respect, the environment representation contains also instances of the `ModuleCache` metaclass. Each instance refers to the modules of a specific kernel version. The execution of the `update_modules` command will update such references as specified in the following. The helper `getModules` is used in the following called rule in order to search for the modules in the location `/lib/modules/version`

```
rule update_modules(version) {
    from
        s : INConfiguration!ModuleCache (
           s.version = version
        )

    to
        t : OUTConfiguration!ModuleCache (
            version <- s.version,
            modules <- thisModule.getModules(version)
        )
}
```

Scrollkeeper

◇ **update_scrollkeeper**, this command updates the model elements which maintain the catalog of the scrollkeeper documents. The metaclasses which are involved in this operation are `SkeeperCatalog` and `SkeeperDocument`. The former maintains the references to all the installed documents, the latter specifies each of them. The semantics of this command is specified in the following called rule. The helper `getSkeeperDocumet` is used to search for new scrollkeeper documents in the system.

```
rule update_scrollkeeper {
    from
        s : INConfiguration!SkeeperCatalog

    to
        t :
            OUConfiguration!SkeeperCatalog
    do {
        t.documents<-thisModule.getSkeeperDocument(s.documents);
    }
}
```

SGML catalog

◇ **postinst_sgmlcatalog**, once SGML documents have be installed the SGML catalog has to be updated. The metaclasses which are involved in this operation are `SGMLCatalog` and `SGMLDocument`. The former maintains the references to all the installed SGML documents, the latter specifies each of them. The semantics of this command is specified in the following called rule. The helper `getSGMLDocumet` is used to search for new SGML documents in the system.

```
rule update_scrollkeeper {
    from
        s : INConfiguration!SGMLCatalog

    to
        t :
            OUConfiguration!SGMLCatalog (
                documents<-thisModule.getSGMLDocument(s.documents)
            )
}
```

◇ **prerm_sgmlcatalog**, this command updates the `SGMLCatalog` which maintains the SGML catalog of the given system. In particular, this command removes all the references to the existing documents even though they are not deleted from the filesystem. This operation is performed by means of the following called rule. The helper `getSGMLCatalog` is used to query the configuration model and retrieve the existing SGML catalog element. Once retrieved, its reference `documents` is nullified.

```
rule prerm_sgmlcatalog{
    using {
        sgmlCatalog : OUTConfiguration!SGMLCatalog = OclUndefined;
    }

    do {
        sgmlCatalog <- thisModule.getSGMLCatalog();
        sgmlCatalog.documents <- OclUndefined;
    }
}
```

- **postrm_sgmlcatalog**, this command removes from the filesystem all the documents of the SGML catalog. In the following rule the helper `isSGMLDocument` is used in order to check if a given file is an SGML document or not.

```
rule postrm_sgmlcatalog {
    from
        s : INConfiguration!File(
            s.isSGMLDocument()
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the File s which is an SGML document is not copied to
        -- the target configuration
    }
}
```

udev

udev provides the link between software commands and hardware devices. As it is such a critical element it has its own set of configuration files to make sure that any new changes in the configuration do not over-write the user provided scripts. Rather than disable the user's preference, new configurations are downloaded but recorded as a new file for the user to merge at a later point.

- **preinst_udev**, this command is for preserving local modifications for reasons related to udev adoption. This command checks to see if the local configuration has been changed by using MD5SUM.

- **postinst_udev**, this command then takes the result of the preinst script and if there has been a change in the local configuration, rather than ovewriting the local configuration which is avoided by package maintainers, instead the new configuration is preserved using a different suffix for the file name.

usrlocal

These scripts pertain to the creation and removal of the /usr/local directory. Maintaining the local configuration files is one of the highest priorities for package maintainers so these scripts relate to preserving any that might exist.

- **postinst_usrlocal**, this command is used to make sure that the creation of the /usr/local directory has only just occurred and was not there before the installation of /usr/local occurred. Adhering to the FileSystem Hierarchy Standard that was last updated in 2004, local preferences and modifications should be stored in this directory. As such the script makes sure to check that the directory did not already exist and therefore possibly be removing the local configurations.

- **prerm_usrlocal**, is a command that checks to see if the /usr/local directory is empty and if so there is no risk of deleting local configuration files. If the directory is non-empty there is a risk of deleting local configurations and hence the command is aborted and the user is informed during the simulation.

Users and Groups

- **user_add(user_name, group_name, homedir)**, this command adds a new user in the configuration. In particular, a new `User` instance is created as specified in the following called rule.

```
rule user_add(user_name, group_name, homedir){
    using {
        newUser : OUTConfiguration!User = OclUndefined;
    }
    do {
        if thisModule.userExists(user_name) then
            OclUndefined;
        else {
            newUser<-thisModule.createUser(user_name);
            newUser.groups <- thisModule.getGroupByName(group_name);
            newUser.home <- thisModule.getLocationByString(homedir);
        }
    }
}
```

The `getGroupByName` and `getLocationByString` helper are used. The former queries the configuration model to retrieve the group which has the name passed as parameter. The latter retrieves from the configuration model the `File` instance which represents the location `homedir`.

◇ **user_remove(user_name,group_name)**, this command is used to remove a user from a group or, if the second parameter is not given, to completely delete the user from the system. This behavior is specified in the following called rules.

```
rule user_remove(user_name){
    from
        s: INConfiguration!User(
            s.name = user_name
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the User s is not copied to the target configuration
    }
}
rule user_remove(user_name,group_name){
    from
        s1: INConfiguration!User,
        s2: INConfiguration!Group (
            s1.name = user_name and s2.name = group_name
        )
    to
        t: OUTConfiguration!User (
            name <- user_name,
            groups <- s1.groups->excluding(s2),
            home <- s1.home
        )
}
```

◇ **group_add(group)**, this command adds a new group in the configuration. In particular, a new `Group` instance is created as specified in the following called rule.

```
rule group_add(group_name){
    using {
        newGroup : OUTConfiguration!Group = OclUndefined;
    }
    do {
        if thisModule.groupExists(group_name) then
            OclUndefined;
        else
            newGroup<-thisModule.createGroup(group_name);
    }
}
```

◇ **group_remove(group_name)**, it removes a given group from the system. The removal can be performed only if no users belong to the group being deleted. This check is performed by means of the helper `isEmpty`.

```
rule group_remove(group_name){
    from
        s: INConfiguration!Group(
            s.name = group_name and s.isEmpty()
        )
    do {
        -- The action block is empty, no action is executed and hence
        -- the Group s is not copied to the target configuration
    }
}
```

Windows manager

- **postinst_wm(wm_location, manual_location)**, this command exploits the `add_alternative` and `inst_doc` commands previously presented in order to add a new alternative for the `x-window-manager` command and a new corresponding manual. In particular the commands will induce the execution of `add_alternative(x-window-manager,wm_location)` and then `inst_doc(x-window-manager, manual_location, '/usr/share/man/man1/x-window-manager.1.gz')`

- **postinst_wm_noman(wm_location)**, if the maintainer does not want to provide the manual of the installed window manager, this command can be used. In this respect, this command is similar to the previous one. The main difference is that it induces the execution of `add_alternative(x-window-manager, wm_location)` and do not install any manual file.

- **prerm_wm(wm_location)**, in order to remove a window manager, it has to be firstly deleted in the alternatives. The execution of this command implies the execution of `rm_alternative(x-window-manager, wm_location)`

Xfonts

- **update_xfonts**, the execution of this command implies the cache update of the installed fonts. In particular, all the `XFontCache` instances have to be updated by considering the fonts contained in the specified locations as described in the following. The helper `createXFonts` is used to search for all the fonts which are installed in the given location and create corresponding `XFont` instances.

```
rule update_xfonts {
    from
        s: INConfiguration!XFontCache
    to
        t : OUTConfigration!XFontCache(
            location <-s.location
            xfonts <- thisModule.createXFonts(s.location)
        )
}
```

Tagging statements

The tagging statements are provided to enable the behavior specification of those scripts which contain commands and statements not covered by the DSL. Since we are interested in reasoning on the configuration model, the provided tagging statements are crucial to still enable the simulation and failure detection possibilities.

The semantics of the tagging statements is quite simple. In fact, tag statements can be considered as atomic actions which change the configuration models. Each tagging statement has a corresponding ATL transformation which can be generally described as follows.

- **add<metaclassName>(<metaclass feature values>)**, this kind of commands are used to add in the configuration new element. For instance, if we consider the metaclass `File`, the command `addFile` is available and maintainers can specify sentences like (`addFile("/home/user/foo",'644')`). By executing this sample statement, a new file named `foo` with '644' as access rules, will be created in the configuration model. In general, given a metaclass `MC` in the configuration metamodel, the corresponding `addMC(feature values)` command has the following semantics

```
rule addMC {
    using {
        newMC : OUTConfiguration!MC = OclUndefined;
    }
    do {
        newMC<-thisModule.createMC(feature values);
    }
}
```

- **delete<metaclassName>(<metaclass feature values>)**, this kind of commands are used to delete elements in the configuration model. In the case of the metaclass `File`, the command `deleteFile` can be used by maintainers to delete files in the abstract representation of the file system. For instance, the command `deleteFile("/home/user/foo")` will delete the file `foo` the directory `/home/user`. Generally speaking, given a metaclass `MC`, the corresponding `deleteMC(feature values)` command is available and it has the following semantics

```
1  rule deleteMC(feature values){
2      from
3          s: INConfiguration!MC(
4              <feature values>
5          )
6      do {
7          -- The action block is empty, no action is executed and hence
8          -- the MC s is not copied to the target configuration
9      }
10 }
```

  Note that the feature values passed to the called rule will be used to select the proper instance of the MC metaclass (see line 4 above) in the source configuration model. In the previous example, the name of the file and its extension will be used to select the right instance of the metaclass `File`.

- **add<metaclassName>_<attribute>(<element>,<attribute value>)**, this kind of command is used to add attribute values for existing elements in the configuration model. For instance, if we consider again the metaclass `File`, the command `addFile_extension("/home/uer/foo","txt")` can be executed in order to specify the extension for the already existing file `/home/user/foo`. In general, given a metaclass `MC` and an attribute `att`, the command `addMC_att(element,attributeValue)` is available and it has the following semantics

```
rule addMC_att(element, attributeValue){
    from
        s: INConfiguration!MC(
            <element>
        )
    to
        t: OUTConfiguration!MC(
            att <- attributeValue
        )
}
```

- **delete<metaclassName>_<attribute>(<element>)**, this kind of command is used to nullify the value for attributes of existing elements in the configuration model. For instance, if we want change the file `/home/user/foo.txt` by deleting the `'txt'` extension, the `deleteFile_extension(/home/user/foo)` command can be used in order to delete the value for the attribute `extension` of the model element which represents the file `/home/user/foo`. In general, given a metaclass `MC` and an attribute `att`, the command `deleteMC_att(element)` is available and it has the following semantics

```
rule addMC_att(element){
    from
```

```
        s: INConfiguration!MC(
            <element>
        )
    to
        t: OUTConfiguration!MC(
            att <- OclUndefined
        )
}
```

- **add<metaclassName>_<reference>(<source element>,<target element>)**, this kind of command is used to establish references between existing elements in the configuration model, with respect to the configuration metamodel. In particular, if we consider the `PackageSetting` metaclass, it has the reference `depends` to refer to other package settings. In this respect, the `addPackageSetting_depends` command can be used in order to establish a reference between two existing package settings. For instance, if `apache2` and `php5` are the package settings of the installed Apache Web server and the PHP5 module, respectively, the execution of the `addPackageSetting_depends(apache2, php5)` command changes the configuration model by adding a reference between the two package settings. In general, given a metaclass `MC` and a reference `ref`, the command `addMC_ref(source, target)` is available and it has the following semantics

```
rule addMC_ref(source,target){
    using {
        sourceMC : OUTConfiguration!MC = OclUndefined;
        targetMC : OUTConfiguration!MC = OclUndefined;
    }
    do {
        sourceMC <- thisModule.getMC(source);
        targetMC <- thisModule.getMC(target);
        sourceMC.ref <- targetMC;
    }
}
```

  The helper `getMC` is used to query the configuration model and retrieve the `source` and `target` elements for updating the reference `ref`.

- **delete<metaclassName>_<reference>(<source element>,<target element>)**, this kind of command is used to nullify a reference between existing elements in the configuration model. For instance, if we consider again the `PackageSetting` metaclass, the `deletePackageSetting_depends` command is available to delete the existing dependency between two existing package settings. If `apache2` and `php5` are the package settings of the installed Apache Web server and the PHP5 module, respectively, and there exists a dependency among them, then the execution of the command `deletePackageSetting_depends(apache2, php5)` changes the configuration model by nullifying such a dependency. In general, given a metaclass `MC` and a reference `ref`, the command `deleteMC_ref(source, target)` is available and it has the following semantics

```
rule deleteMC_ref(source, target){
    from
        s: INConfiguration!MC(
            s = thisModule.getMC(source)
        )
    to
        t: OUTConfiguration!MC(
            ref <- s.ref->excluding(thisModule.getMC(target))
        )
}
```

Chapter 5

Sample DSL applications

In this chapter we look at how to counter-act some of the failure types identified in Chapter 2 by using a model based approach that was explained in Chapter 3 and described in detail through the creation of a DSL in Chapter 4. The creation of a DSL had a specific purpose in mind which was to provide a mechanism in which we could identify and possibly fix invalid configuration states for packaging systems. We must identify that the DSL is an evolutionary language that can incorporate changes as they are found. The reasons why we selected this approach have been discussed in depth in Chapter 3 and also in the deliverable D2.1.

Current meta-installers are capable of detecting a sub-set of the configuration failures which are static failures and are identified manually or using helper utilities such as *deb_helper* and *rpm-helper*. Once package maintainers know that there is a problem with the interoperability of their package and another, they can use meta-description tools to encode information such as creating conflicts and other meta-info to suggest to the meta-installer when a possible problem might occur. There are however times when such meta-info is insufficient as it is not expressive enough to detect certain types of known errors.

In this chapter we discuss how the proposed DSL and the model driven approach can be used to support some representatives of the upgrade failures identified in Chapter 2.

5.1 Apache2 and libapache-mod-php5

In this section we describe how the proposed simulation approach and how the DSL can be used to discover failures. These failures can occur when the maintainer scripts of the `libapache-mod-php5` package are not complete and are missing some statements which are required to obtain a valid target configuration. The problem we discovered is that the *prerm* script of `libapache-mod-php5` does not disable the reference to the `PHP5` module in `Apache2` prior to the removal of `PHP5`. The problem can be detected once the target configuration is reached and the transformation vector to the model is evaluated to check if some of the possible failures identified in Chapter 2 occur.

As said in the previous chapters, a system configuration is composed of artifacts necessary to make computer systems perform their intended functions. In this respect, the *Configuration* metamodel has been provided to specify the main concepts which make up the configuration of a FOSS system. In particular, the `Environment` metaclass enables the specification of loaded modules, shared libraries, and running process as in the sample configuration reported in Figure 5.1. In such a model the reported environment is composed of the services `apache2`, and `sendmail` (see the instances `s1` and `s2`) corresponding respectively to the running web and mail servers.

All the services provided by a system can be used once the corresponding packages have been installed and have a properly configured (`PackageSetting`). Moreover, the configuration of an installed package might depend on other package configurations.

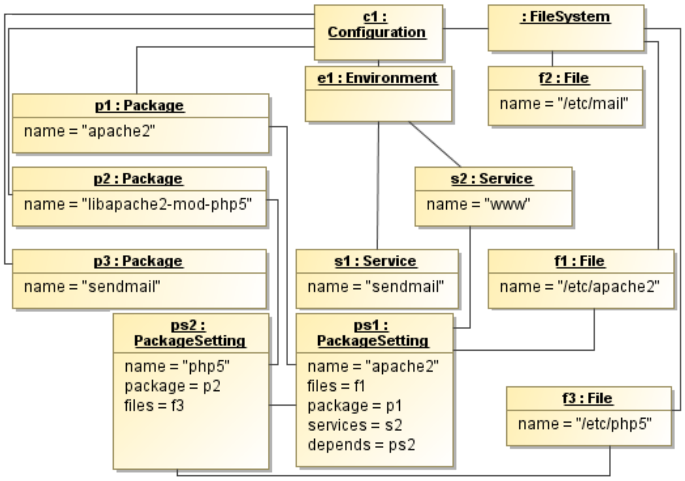


Figure 5.1: Sample configuration model

Considering the case of the PHP5 upgrade, the instances `ps1` and `ps2` of the `PackageSetting` metaclass in Figure 4.6 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 Figure 5.1) and both are also associated with the corresponding files which store their configurations.

Note that at the level of inter-package relationships 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 the proposed metamodels which embody domain concepts which are not taken into account by current package manager tools.

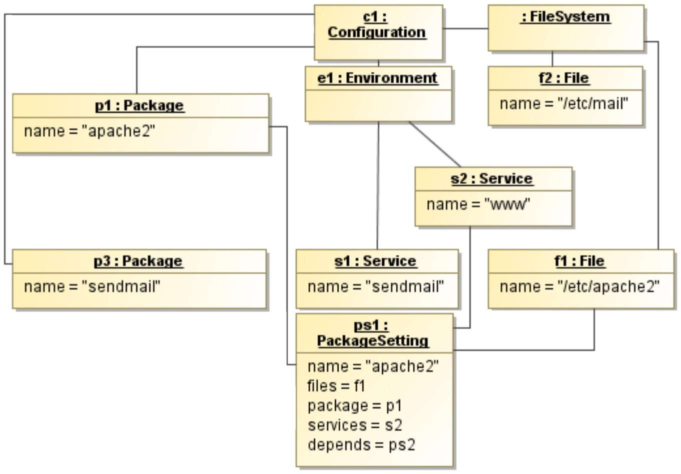


Figure 5.2: Not valid configuration model after `libapache-mod-php5` removal

Figure 5.2 reports the configuration model which is obtained after the removal of the `libapache-mod-php5` package supposing that its *prerm* scripts does not disable `PHP5` in `Apache2`. The problem in such a model is that the package setting of the Web server still depends on `PHP5` even though such a module is not available in the file system.

Once the problem has been detected, maintainers can solve the problem by adding the following script as *prerm* of `libapache-mod-php5`

```
#<%
delPackageSetting_dependences(apache2,php5);
#%if [-e /etc/apache2/apache2.conf]; then
```

```
#%     a2dismod php5 || true
#%fi
#%>
```

In particular, the dependency between the package settings of `Apache2` and `PHP5` has to be deleted in order to lead to a valid configuration. In this respect, the `delPackageSetting_dependences` DSL tagging command has been used to wrap the corresponding script which will be executed on the real system once the simulation phase has success.

5.2 A sample use-before-define failure: Freeradius-2.1.3

In this section we discuss the adoption of the proposed approach to detect a sample *use-before-define* failure. In this respect, we will consider the `Freeradius-2.1.3` package which is affected by this kind of problem. In particular, as described in Section 2.3, the package `Freeradius-2.1.3` depends on the `freeradius-libs`. If the library is installed before the `freeradius` package then it would expect a user to be present that is only created by the configuration file in the freeradius package. Although the package maintainer has correctly identified that `freeradius` depends on `freeradius-libs` the package maintainer cannot then add a dependency from `freeradius-libs` back without creating a nested loop.

The information contained in the configuration model, is enough to detect this kind of failure even during upgrade simulations. In fact, according to the configuration metamodel presented in Chapter 4, the `User` and `Group` metaclasses are provided to model the users and the groups of a given system.

Before accessing the elements of a configuration model, the simulator checks for their existence and if they are not available, the simulation stops and the user is informed about the problem. In the `freeradius` example the `freeradius-libs` installation does not fail if the `radius` user and group are already in the system otherwise the installation fails. Vice-versa, if the installation of `freeradius` is performed first, the configuration model changes since the following scripts is executed

```
#<%
addGroup(radius);
addUser(radius,radius);
#% getent group radiusd >/dev/null || /usr/sbin/groupadd -r -g 95 radiusd
#% getent passwd radiusd >/dev/null || /usr/sbin/useradd -r -g radiusd -u 95 -c "radiusd user" -s /sbin/nologin radiusd > /dev/null 2>&1
#%>
```

In particular, new instances of the `User` and `Group` metaclass are added and the simulator can check their existence to install `freeradius-libs` without problem. The proposed solution is able to detect such problems even if users adopt meta-installers which cannot resolve the circular dependency.

By using the general approach of checking the metaclasses when the configuration script looks for commonly referred data and seeing if it is present, we can stop the simulation and inform the user before any actual installation has occurred. Currently, depending on the meta-installer, the installation may or may not fail. If it does not fail then the fall-back mechanism as used by RPM is to use root for the User and Group IDs (UID, GID). The installation will succeed in this case but the user and group is not as specified in the configuration file. In this case a "not valid" configuration has been reached. What is particularly dangerous about this is the potential security problems as there is little or no warning except in the log files that any deviation from that outlined by the script has happened. By using the DSL on the other hand we are able to detect this configuration problem and stop the installation from occurring. Further modification of the simulator and the DSL will allow for the automatic recovery of this type of error if another package contained in the manifest has a configuration script that would set up these values.

Chapter 6

Conclusion

In this deliverable we presented a DSL to specify the behavior of maintainer scripts and to predict several of their effects on package upgrades. In fact, the present generation of package managers only rely on package meta-information which is not expressive enough to predict, detect, and eventually manage upgrade failures which can occur because of erroneous or incomplete scripts.

The model-driven approach outlined in this Deliverable rely on maintaining a model-based description of the system and simulating upgrades in advance on top of it, to detect predictable upgrade failures and notify the user before the actual installation occurs and 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 missing sub-components), which are currently not expressible as inter-package relationships.

Going into more detail, the main contributions of this deliverable are:

1. we have proposed a classification of common failures which can occur during the case of system upgrades (see Chapter 2);
2. we have defined a DSL for specifying maintainer scripts with respect to the upgrade failure classification (see Chapter 4);

By developing the DSL we have provided a sound framework from which to develop tools that will be able to track changes and merge with the results of Work Packages 2 and 4. This is also the first milestone of the Work Package, M3.1 and provides a basis from which the other Work Packages within the MANCOOSI project can rely on.

Concerning contribution 1, currently only a small sub-set of the failures identified in Chapter 2 are being detected and even then they are related to dynamic failures only. With a lack of quantative analysis on the subject we had to suggest certain potential failures through experience of distribution of a system.

As we have mentioned before, the approach outlined in Chapter 3 is designed to be performed using an evolving language, the DSL. If new failure types are identified then through a combination of using the tagging system and viewing failures that have not been detected by either the simulator or failure detector we will refine the DSL to incorporate these identified failure types. Through the refinement of the model using injection mechanisms that will be specified completely in Deliverable 2.3 we are able to increase the number of failures that we are able to detect, so even if the initial set of failures seems retrospectively quite limited we provide a mechanism in which to incorporate changes.

This is in contrast to the static nature of dependencies that are written on the update of a package by an individual or set of maintainers but that do not reflect the changes of the package universe. The model proposed and described in Chapter 4 will take an overview of the package universe and as such will be able to detect a set of failures that are currently very difficult or impossible to detect using individual meta-information stored in packages and the current generation of meta-installers.

By considering the identified failures, and the proposed DSL we briefly discussed the application of our approach to deal with some types of package upgrade failures. In particular, we considered a sample failure which involved the Apache Web server and the PHP5 module. This is an example of an incomplete set of configuration files. Although each of the configuration files can be run and do not have any syntactical failures the overall result of installing and removing the package in question, leaves certain configurations in situ. These changes that were introduced by the installer maintainer script are not fully removed and this has the result in leaving the configuration in an undefined state.

This failure as identified in Chapter 2 is a “slow failure” in that it may not be detected instantly by a user of the system and can take an indeterminate time before it is discovered or has a consequence on the operation of part of the system. During this time other packages may be installed, removed, upgraded and only when the application uses the erroneous configuration that the problem occurs. It may be very difficult to track down this type of failure as a long time may have elapsed between the upgrade of the local machine’s packages and the current error. This is where the approach of using a model of the system can help assist in finding the problems.

In the first instance the simulator may be able to pick up this sort of failure as indicated in Chapter 5 and warn the user of the potential problem. If the failure is not detected at this stage the transactional log that stores the changes to package configurations can be analysed to see what may have affected the system.

If these mechanisms fail or otherwise it may be important to revert the changes performed by the simulator and the actual system. By having stored the selection of the mechanism of what choice was made in the case of a 1-many choice as it happened in the transaction log it will be possible to indicate what the reverse action might be.

Instead of having to perform a complete state refresh and going back to a saved configuration like certain systems identified in Chapter 1 instead it will be possible to roll-back an individual package to a certain state. In the case of the identified example the meta-class will record the changed variables and the simulator will notice that a value has had a state changed but not reverted. For trivial commands that have one to one mappings it may be possible to even resolve these problems using the simulator. If unable to solve the failure case it hopefully will be able to at least identify the problem and prompt the user for action.

We have considered another example related to the use- `use-before-define` failure type. It is a failure that occurs when a maintainer script refers to a variable in the system that hasn’t been set up yet but would be provided in a transaction set. It differs from a missing resource in that another package would set up the missing variable but because of the order the transaction set is handled, the definition occurs after the first use. This is a new term as described in 2 and is derived from programming languages. The problem is a dormant, widespread in GNU/Linux systems in that there are missing dependecies but because the current-breed of meta-installers order the packages in a fairly uniform method that these failures never come to light. In the scope of the MANCOOSI project where we will be looking at modifying the ordering of the packages as well as generating tools to decide which packages to install to meet various criteria it would inevitably have uncovered this sort of problem. By identifying the problem at this stage, the DSL can help track down what are generally seen as more subtle failures as well as allowing the tools that will be generated in Work Package 4 to modify the ordering and transaction set by uncovering possible failures using the simulator.

During our research we also examined one of the potential benefits of a DSL. Through the research we have postulated that there are likely many more positive advantages provided by the DSL than the types of failure case that it will identify. One generic type of advantage we explored was that of the cache-rebuilders and in particular ldconfig. From the research carried out it was apparent that a lot of the maintainer scripts had lots of elements in common and most of the frequently shared templates related to cache-rebuilding of some sort or another. The process is quite similar for most of the scripts in that they would copy a file to the default directory and then call the associated cache-rebuilder. The issue of optimising the number of times a cache-rebuilder has been addressed before but one of the advantages of a system that overviews package meta-installers is that it can perform analysis on the maintainer scripts and minimise the number of times a cache-rebuilder has to be called. This is not the main reason that a DSL was considered but as one of the advantages brought forward it helps the overall aim of managing evolving, complex, packaging systems and could possibly reduce the amount of information that would need to be stored as meta-data within current packages.

The proposed DSL and the overall simulation approach will be integrated with tools produced in later tasks in this Work Package, which are able to keep track of the changes which occur on real system configurations, in order to be able to roll them back, restoring previous safe system states.

Part A

Fragment of the MANCOOSI metamodel

The metaclasses which have been considered for the DSL semantic specification are reported in the following. They are given in KM3 which is a lightweight textual metamodel definition language allowing easy creation and modification of metamodels [1]. KM3 is based on the same core concepts used in OMG/MOF and EMF/Ecore 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 graphical representation of the full MANCOOSI metamodel is available at `http://www.mancoosi.org`.

```
-- Description of settings, environment, files, services, devices that can be modified by means script statements, set of modules or shared libraries available in a given configuration
    class Environment {
        reference runningServices[0-*] : Service oppositeOf env;
        reference alternatives[0-*] container : Alternative oppositeOf env;
        reference users[1-*] container : User oppositeOf env;
        reference groups[1-*] container : Group oppositeOf env;
        reference emacspkg container : EmacsPackage oppositeOf env;
        reference iconCache container : IconCache oppositeOf env ;
        reference desktopDb container : DesktopDB oppositeOf env;
        reference mimeTypeHandlerCache container : MimeTypeHandlerCache oppositeOf env;
        reference libraryCache[1-1] container : LibraryCache oppositeOf env;
        reference skeeperCatalog container : SkeeperCatalog oppositeOf env;
        reference sgmlCatalog[1-1] container : SGMLCatalog oppositeOf env;
        reference configuration : Configuration oppositeOf environment;
        reference moduleCache[1-*] container : ModuleCache oppositeOf env;
        reference xfontCaches[1-*]  container : XFontCache oppositeOf env;
        reference gconf container : GConf oppositeOf env;
        reference menu container : Menu oppositeOf env;
        reference init container : Boot oppositeOf env;
    }

    class FileSystem {
        reference root container : File oppositeOf fs;
        reference configuration : Configuration oppositeOf fileSystem;
    }

    -- This class is defined to model the services which has to be started when the system is started
    class GConf {
        reference confFiles[0-*] : File;
        reference schemas[0-*] : File;
        reference env : Environment oppositeOf gconf;
    }

```

[1] `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/KM3-FMOODS06.pdf`

```

    -- This metaclass models the catalog of all the possible menu entries which might be intalled
    class ApplicationMenuCatalog  {
        attribute executable : String;
        reference menu : Menu oppositeOf catalog;
    }

    class Menu {
        reference entries[0-*] container : MenuEntry oppositeOf menu;
        reference catalog container : ApplicationMenuCatalog oppositeOf menu;
        reference env: Environment oppositeOf menu;
    }

    class MenuEntry {
        reference menu : Menu oppositeOf entries;
        reference executable : File;
        reference parent : MenuEntry;
    }

    class Boot {
        reference services[1-*] : Service;
        reference env : Environment oppositeOf init;
    }

    class Service {
        reference executable : File;
        reference env: Environment oppositeOf runningServices;
    }

    class File {
        attribute extension : String;
        attribute description: String;
        attribute size: Integer;
        attribute checkSum: String;
        attribute isDirectory : Boolean;
        attribute suid : Boolean;
        attribute giud : Boolean;
        attribute permission : String; -- This string will contain the permission policy of files in the classical UNIX form
        attribute location : String;
        reference fs: FileSystem oppositeOf root;
        reference childs[0-*] container : File oppositeOf parent;
        reference parent : File oppositeOf childs;
    }

    class DocumentationFile extends File {
        reference pkg : Package oppositeOf documentationFiles;
    }

    class InformationFile extends File {
    }

    class Alternative  {
        reference current : File;
        reference location : File;
        reference env : Environment oppositeOf alternatives;
    }

    class PackageSetting  {
        reference services[0-*] : Service;
        reference files[0-*] : File;
        reference configuration : Configuration oppositeOf packageSettings;
        reference pkg : "Package";
        reference dependences[0-*] : PackageSetting;
    }

    class IconCache {
        attribute mtime : String;
        reference env : Environment oppositeOf iconCache;
        reference icons : File;
    }

    class DesktopDB {
        reference env : Environment oppositeOf desktopDb;
        reference applications : File;
    }


    class MimeTypeHandlerCache {
        reference env : Environment oppositeOf mimeTypeHandlerCache;
        reference handlers[0-*] : MimeTypeHandler;
    }

    class MimeTypeHandler {
        reference type : String;
        reference handler : File; -- refer to the executable file
    }


    class XFontCache {
        reference xfonts[1-*] container : XFont oppositeOf xfontCache;
        attribute location : String;
        reference env : Environment oppositeOf xfontCaches;
    }

    class XFont  {
        reference xfontCache : XFontCache oppositeOf xfonts;
        reference file[1-*] : File;
    }

    -- This class models the typical /etc/ld.so.conf content and the found shared
        ↪libraries in the referred locations
    class LibraryCache {
        reference locations[0-*] : File;
        reference sharedLibraries[0-*] container : SharedLibrary oppositeOf libraryCache
            ↪;
        reference env : Environment oppositeOf libraryCache;
    }

    class SharedLibrary {
        attribute name : String;
        reference file : File;
        attribute version : String;
        reference libraryCache: LibraryCache oppositeOf sharedLibraries;
    }

    class ModuleCache {
        attribute version : String;
        reference modules[0-*] container : Module oppositeOf moduleCache;
        reference env : Environment oppositeOf moduleCache;
    }

    class Module  {
        reference file : File;
        reference moduleCache: ModuleCache oppositeOf modules;
    }

    -- This metaclass is used to specify SGML catalogs
    class SGMLCatalog  {
        reference env : Environment oppositeOf sgmlCatalog;
        reference documents[0-*] : SGMLDocument;
    }

    class SGMLDocument {
        reference location : File;
        reference document : File;
    }

    -- This metaclass is used to specify Scrollkeeper catalogs
    class SkeeperCatalog  {
        reference env : Environment oppositeOf skeeperCatalog;
        reference documents[0-*] : SkeeperDocument;
    }

    class SkeeperDocument {
        --reference location : File;
        --reference document : File;
    }

    -- Description of EmacsPackage
    class EmacsPackage  {
        reference files[1-*] container : File;
        reference env : Environment oppositeOf emacspkg;
    }
    -- End of Description of EmacsPackage


    -- Description of User and Group concepts
    class User  {
        reference env : Environment oppositeOf users;
        reference groups[1-*] : Group oppositeOf users;
        reference home : File;
    }

    class Group  {
        reference env : Environment oppositeOf groups;
        reference users[0-*] : User oppositeOf groups;
    }
    -- End of Description of User and Group concepts

-- End Description of settings, environment, files, services, devices
-- that can be modified by means script statements
```

Bibliography

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

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

[BÖ2] E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.

[B05́] J. Bézivin. On the Unification Power of Models. *SOSYM*, 4(2):171–188, 2005.

[BCvH$^+$03] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *ICATPN*, pages 483–505, 2003.

[Béz05] Jean Bézivin. On the Unification Power of Models. *J. Software and Systems Modeling*, 4(2):171–188, 2005.

[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.

[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.

[CGI$^+$08] Betty H. C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogerio de Lemos et al. 08031 – software engineering for self-adaptive systems: A research road map. In Betty H. C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, number 08031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

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

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

[DH07] Eelco Dolstra and Armijn Hemel. Purely functional system configuration management. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

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

[ea06] Linda Northrop et al. *Ultra-Large-Scale Systems - The Software Challenge of the Future.* SEI Institute, Carnegie Mellon University, 2006.

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

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

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

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

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

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2005.

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

[Kle07] A.G. Kleppe. A language description is more than a metamodel. 2007.

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

[MBdC$^+$06] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. *International Conference on Automated Software Engineering*, pages 199–208, 2006.

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

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

[Obj03a] 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.

[Obj03b] Object Management Group (OMG). UML 2.0 Infrastructure Final Adopted Specification, 2003. OMG document ptc/03-09-15.

[Obj03c] Object Management Group (OMG). XMI 2.0 XML Metadata Interchange, 2003. OMG document formal/2003-05-02.

[OMG02] OMG. MOF 2.0 Query/Views/Transformation RFP, 2002. OMG document ad/2002-04-10.

[OMG03] OMG. MDA Guide version 1.0.1, 2003. OMG Document: omg/2003-06-01.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

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

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

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

[Smi00] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[The09a] The Free Software Foundation. Bash shell. `http://www.gnu.org/software/bash/`, 2009.

[The09b] The Perl Foundation. The perl directory. `http://www.perl.org/`, 2009.

[TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *SPLC*, volume 3714 of *LNCS*, pages 198–209. Springer, Oct 2005.

[TZ08] Ralf Treinen and Stefano Zacchiroli. Description of the CUDF format. Mancoosi project deliverable D5.1, November 2008.

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

[Wor] World Wide Web Consortium (W3C). Web Ontology Language (OWL). http://www.w3.org/2004/OWL.

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

[ZX04] Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, 39(3):14–30, 2004.