Model Transformation for Verification: building the basis for a generic tool

Bachelor thesis presented to
CUI
University of Geneva

by

Stephane Heck

Supervisor:
Prof. Dr. Didier Buchs

Co-Supervisor:
Luis Pedro

12 September 2005
Acknowledgements

This bachelor thesis is dedicated to Stephanie, Mum, the Heck and Boehi families, my friends and all the people who have been supporting me during my study of Computer Science at the University of Geneva.
It is also dedicated to Luis, Ang, Adrien, Sergio, the Software Modeling and Verification (SMV) Group and bachelor students for their valuable help and great friendship during the bachelor projects.
A special thank to Alex, from Down Under, for his help and encouragement during the writing of this thesis.
Abstract

This report presents the bachelor project called **Metamodel Transformation for Verification (MTV): building the basis for a generic tool, under the supervision** of Software Modeling and Verification (SMV) group in the University of Geneva. The document exposes an introduction to Model Driven Architecture (MDA) and Meta Object Facility (MOF), and it gives a survey of the software tools and concepts used for the projects realization.

It also provides a description all project phases of the software project, in particular, the Use Cases, Story Boards, Architecture and Implementation. It also shows a case study for Concurrent Object Oriented Petri Nets (COOPN).

The bachelor project proposed by the SMV group, to which this document concerns, was defined as the analysis and development of a software tool that could be the basis of a much more general and featured tool that will encapsulate both generic and specialized model browsing for different languages. This tool was developed from the very beginning and its architecture and implementation were driven by the different use cases and user requirements collected in the first phases of the project. The resulting application is a combination of back-end functionalities, that include integration of other Application Program Interfaces (APIs), model exploration algorithms, as well as a an intuitive and user friendly Graphical User Interface (GUI).
Contents

Acknowledgements ii
Abstract iii

1 Introduction 8
  1.1 MDA Model Driven Architecture ....................... 8
  1.2 MOF Meta Object Facility ............................. 11
  1.3 MOF metadata architecture ........................... 13
    1.3.1 Relation between source code and model .......... 14
    1.3.2 Relation between model and metamodel .......... 15
    1.3.3 Relation between metamodel and meta metamodel .. 21
  1.4 Transformation using MDA ............................ 30

2 Description of the project 32

3 Analysis of the problem 34
  3.1 Overview of project phases .......................... 34
  3.2 Survey of software tools used in this area .......... 36
    3.2.1 XMI XML Metadata Interchange .................. 36
    3.2.2 JMI Java Metadata Interface ..................... 37
    3.2.3 MDR Meta Data Repository ....................... 41
    3.2.4 UML2MOF ........................................ 43

4 Use Cases 45
  4.1 Primary and Secondary Actors ....................... 45
  4.2 Create a project in the MTV ......................... 45
  4.3 Open a project in the MTV ........................... 47
  4.4 Update metamodel in the MTV ....................... 48
4.5 Update JMI in the MTV ................................. 50
4.6 Browse metamodel in the MTV ....................... 51
4.7 Browse a model in the MTV ............................ 52
4.8 Upload a model in the MTV ............................. 52
4.9 Diagram ............................................. 54

5 Story Boards ........................................... 55
5.1 The MTV graphical user interface ..................... 55
5.2 Create a project in the MTV ............................ 59
5.3 Open a project in the MTV .............................. 60
5.4 Update metamodel in the MTV ......................... 63
5.5 Update JMI in the MTV ................................ 64
5.6 Browse a metamodel in the MTV ....................... 67
5.7 Browse JMI in the MTV ................................ 69
5.8 Browse a model in the MTV ............................. 71
5.9 Upload a model in the MTV ............................. 72

6 Architecture .......................................... 75
6.1 MTV Architecture ...................................... 75
6.2 MTV packages overview ................................ 77
6.3 External APIs .......................................... 81
   6.3.1 JMI API .......................................... 81
   6.3.2 NetBeans MDR API ................................ 83
   6.3.3 NetBeans UML2MOF Tool ......................... 83
   6.3.4 Ant API .......................................... 84
   6.3.5 JMI inheritences in MTV framework ............. 84

7 Implementation ........................................ 85
7.1 Project management .................................... 85
   7.1.1 Project Name ..................................... 86
   7.1.2 Getting information from existing projects ...... 86
   7.1.3 Creating the structure of a new Project .......... 86
   7.1.4 Browsing Project Settings and Resources ......... 87
   7.1.5 Classes involved in project management ......... 88
7.2 MDR repository management ........................... 89
   7.2.1 Using NetBeans UML2MOF Tool ................. 89
   7.2.2 XMI[UML] to XMI[MOF] transformation .......... 90
   7.2.3 Load a metamodel in MDR repository .......... 91
## List of Figures

1.1 Relations between CIM, PIM, PSM and code .................................. 10
1.2 MOF metadata architecture ...................................................... 13
1.3 The car control system class diagram ........................................ 14
1.4 Relation between model and metamodel: class instances ............. 16
1.5 Relation between model and metamodel: attribute instances ........ 17
1.6 Relation between model and metamodel: operation instances ...... 18
1.7 Relation between model and metamodel: datatype instances ...... 19
1.8 Relation between model and metamodel: association instances .... 20
1.9 Relation between model and metamodel: association-end instances ... 21
1.10 Relation between model and metamodel: generalization instances ... 22
1.11 Relation between metamodel and meta metamodel: class instances ... 23
1.12 Relation between metamodel and meta metamodel: attribute instances ... 24
1.13 Relation between metamodel and meta metamodel: datatype instances ... 25
1.14 Relation between metamodel and meta metamodel: association instances ... 26
1.15 Relation between metamodel and meta metamodel: association-end instances ... 27
1.16 Relation between metamodel and meta metamodel: generalization instances ... 28
1.17 MOF and metamodels in MDA transformations ......................... 30
1.18 P*M to P*M Transformation ................................................. 31
3.1 Waterfall process for MTV ..................................................... 35
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>XMI: standard mechanism for tools</td>
</tr>
<tr>
<td>3.3</td>
<td>JMI reflective interfaces</td>
</tr>
<tr>
<td>3.4</td>
<td>Relation between JMI reflective and JMI specific interfaces from a given metamodel</td>
</tr>
<tr>
<td>3.5</td>
<td>Structure of the MDR project (source MDR website)</td>
</tr>
<tr>
<td>3.6</td>
<td>UML2MOF transformation</td>
</tr>
<tr>
<td>4.1</td>
<td>Use Case Diagram of the MetaModelTool</td>
</tr>
<tr>
<td>5.1</td>
<td>Welcome panel</td>
</tr>
<tr>
<td>5.2</td>
<td>File menu</td>
</tr>
<tr>
<td>5.3</td>
<td>MDR settings menu</td>
</tr>
<tr>
<td>5.4</td>
<td>Help menu</td>
</tr>
<tr>
<td>5.5</td>
<td>Metamodel panel</td>
</tr>
<tr>
<td>5.6</td>
<td>Model panel</td>
</tr>
<tr>
<td>5.7</td>
<td>The user selects the create project option</td>
</tr>
<tr>
<td>5.8</td>
<td>The user selects a XMI[UML] file.</td>
</tr>
<tr>
<td>5.9</td>
<td>The system displays the metamodel panel</td>
</tr>
<tr>
<td>5.10</td>
<td>The user selects the open project option</td>
</tr>
<tr>
<td>5.11</td>
<td>The user selects the project to open</td>
</tr>
<tr>
<td>5.12</td>
<td>The system opens the project and displays the metamodel panel</td>
</tr>
<tr>
<td>5.13</td>
<td>The user selects the update metamodel option</td>
</tr>
<tr>
<td>5.14</td>
<td>The system displays a warning message</td>
</tr>
<tr>
<td>5.15</td>
<td>The user selects a XMI[UML] file</td>
</tr>
<tr>
<td>5.16</td>
<td>The system displays the metamodel panel</td>
</tr>
<tr>
<td>5.17</td>
<td>The user selects the update JMI option</td>
</tr>
<tr>
<td>5.18</td>
<td>The system displays a warning message</td>
</tr>
<tr>
<td>5.19</td>
<td>The system overrides JMI and displays the metamodel panel</td>
</tr>
<tr>
<td>5.20</td>
<td>The user selects the browse metamodel option</td>
</tr>
<tr>
<td>5.21</td>
<td>The system displays a diagram oriented view of the metamodel</td>
</tr>
<tr>
<td>5.22</td>
<td>The user changes the view of the metamodel</td>
</tr>
<tr>
<td>5.23</td>
<td>The system displays a package oriented view of the metamodel</td>
</tr>
<tr>
<td>5.24</td>
<td>The user selects the option to browse the JMI</td>
</tr>
<tr>
<td>5.25</td>
<td>The system displays a tree representation of the JMI</td>
</tr>
<tr>
<td>5.26</td>
<td>The user selects the model to display and the browse model option</td>
</tr>
<tr>
<td>5.27</td>
<td>The system display a diagram oriented of the model</td>
</tr>
<tr>
<td>5.28</td>
<td>The user selects the option to upload a model</td>
</tr>
</tbody>
</table>
5.29 The user selects the model to upload .......................... 74
5.30 The model has been uploaded in the project .................... 74

6.1 responsibility based layering strategy ............................. 76
6.2 MTV package dependencies .......................................... 77
6.3 gui.metaModel package overview .................................... 79
6.4 gui.model package overview ........................................... 80
6.5 gui.bars package overview ............................................. 81
6.6 gui.logPanel package overview ....................................... 82
6.7 MTV framework with external APIs ................................. 82
6.8 JMI inheritance in MTV framework ................................. 84

7.1 project package class diagram ........................................ 88
7.2 mdrControl package overview ....................................... 93

8.1 reduced and simplified COOPN metamodel ...................... 103
8.2 Package oriented view of the COOPN metamodel ............... 109
8.3 diagram oriented view of the drink vending machine .......... 111
Chapter 1

Introduction

This chapter presents an overview of Model Driven Architecture (MDA) and its concepts of Platform Independent Model (PIM), Platform Specific Model (PSM), and another important standard of the Object Management Group (OMG) called Meta Object Facility (MOF).

1.1 MDA Model Driven Architecture

There is no doubt that models are an essential feature in software development. Models are useful communication tools for people involved in the project development process: project managers and software developers. They also provide the possibility of representing concrete systems with different levels of abstraction.

The main objective of MDA is to use models for each phase in the software development process. This approach specifies different levels of abstraction in order to manage complexity in systems. High abstract models represent general application architecture, while concrete models specify communication protocols or algorithms.

The use of models in the development process offers several advantages. However, the coherence between models describing the application must be guaranteed. This coherence depends on the traceability between models, or in other words, the ability to follow the life of the development process, both forward and backward. The purpose of MDA is to use model transformation
1.1. MDA Model Driven Architecture

for a better model reliability.

MDA defines guidelines for structuring specifications expressed as models. It separates the specification of system functionality both from the specification of the implementation and from the functionality on a specific platform technology. It recommends using Computation Independent Model (CIM), PIM and PSM to develop an application [1]. The following items provides a description of these three models.

CIM Computation Independent Model

The first task when building a new application is to specify the client requirements. The objective is to create a requirement model for the new application. Such a model should present an application environment in order to define functionalities provided by the application and define external entities which the system will interact with. For example, when using Unified Modeling Language (UML), a requirement model is expressed with a Use Case diagram.

PIM Platform Independent Model

PIM is a model at a high level of abstraction. It expresses logic functionality and behaviour of the system, undistorted by technology. The goal of PIM is to have a model that can be used for several platforms. This model can then be transformed to various specific platforms. Usually, PIMs are generated during analysis or conception phases in the development process.

PIMs are the bases for all MDA transformations processes. Thus, PIMs must be particularly well described to allow automatic transformations.

PSM Platform Specific Model

A PIM can be transformed into several PSM, according to the platform technology. PSM is the source for automatic code generation. It contains all information relative to an execution platform, such as file systems and identification process.
For instance, a PSM is an UML profile, which is an adaptation of UML for a specific platform, like UML profile for Enterprise Java Bean (EJB).

The transformation between models at a certain abstraction level to another abstraction level is defined by MDA as follow:

**Refinement** stands for generating a more concrete model from an abstract model.

**Abstraction** stands for the process of removing details from a model to generate a more abstract model.

**Mapping** stands for the process transforming a model to another at the same abstraction level.

Figure 1.1 illustrates the relations between CIM, PIM, PSM and code, identifying all possible transformation types among them.

*Figure 1.1. Relations between CIM, PIM, PSM and code*
1.2 MOF Meta Object Facility

As already mentioned in section 1.1 MDA is based on models for all phases of the software development cycle. Such models include CIM, PIM or PSM. Each model is specified in a particular formalism that can be described with a metamodel. MDA has to define a standard modelisation formalism that can support all metamodel based formalism. This standard is the meta metamodel, called Meta Object Facility (MOF).

MOF terminology includes the concepts of metadata, metamodel and metaobject described in the following items.

Metadata

The data is the information that is stored by a computer. It also means information. The term metadata is used to refer data whose purpose is to describe other data [2].

Metamodel

A model describes a system using a well-defined language. The metamodel of a language, also known as abstract syntax, is a description of all the concepts that can be used in that language [3]. It is important to say that a metamodel does not describe a language in terms of semantics.

Another important point in MOF terminology is the extension of the meaning of instantiation. In MOF, a model is an instance of a metamodel and a metamodel is an instance of a meta metamodel, etc.

Metaobject

In UML, an object is an instance of a class. As an analogy, the term metaobject is used to refer to an abstract or technology specific object that represents metadata. [2]

MOF Specification does not define any standard to graphically represent MOF. However a mapping between MOF and UML is defined. Therefore, UML can be used to define metamodels that are instances of MOF. Similarly, no concrete syntax is defined in the MOF Specifications to describe
MOF. For instance, the XML Metadata Interchange (XMI) format (see section 3.2.1) defines a concrete syntax for MOF.

Another important feature of MOF is the fact that this language is self-described, in other words, MOF can describe a MOF model.

MOF defines abstract syntaxes for several languages. Examples of MOF metamodels are the existing definitions of abstract syntax for: UML, Common Warehouse Metamodel (CWM), Extended Markup Language (XML) and finally MOF itself. All these standards are defined by the OMG. The most important feature of MOF is the ability to describe any metamodel.

The following section introduces the MOF metadata architecture, a layering framework based on models, metamodels and meta metamodels.
1.3 MOF metadata architecture

MOF architecture is based on the four layer metadata architecture. Figure 1.2 shows an example of the MOF metadata architecture.

The M0 level (data layer) contains the data to describe. In the example above, the data layer contains source code in different programming languages, such as Java or C++, and also data in XML format.

The M1 level (model layer) describes the information in the data layer (M0 level). It contains all models from which these applications are generated. Please note that a model in M1 level can be transformed into source code in several programming languages, depending if the target language supports all the functionalities of what is represented by the model.

The M2 level (metamodel layer) describes the models in the M1 level. In this example, it contains the UML metamodel, the COOPN metamodel and
Chapter 1. Introduction

The M3 level (meta metamodel layer) describes the metamodels in the M2 level. The meta metamodel is MOF. Using this methodology, higher levels are not necessary because MOF is self-described.

1.3.1 Relation between source code and model

This section describes relations between source code and model. This can be illustrated by the following example: an embed system in a car that allows it to move provided that there is a driver and the four tyres are inflated. In terms of software development, the system might be represented with an UML class diagram. Figure 1.3 illustrates the class diagram of this car control system.

Figure 1.3. The car control system class diagram

Once the model has been described, the source code can be automatically generated. However, some implementation points are not defined in the model. The following code sample provides an example where the developer has decided to use a `Vector` as a data structure to store the four wheels of the car. The difference between the model shown in 1.3 and this source code illustrates a typical example of an abstraction, where technical
1.3. MOF metadata architecture

considerations are omitted in models.

Listing 1.1. Car.java

```java
public class Car extends Vehicle {
    public Vector wheels;
    protected Person driver;

    public Car() {
        wheels = new Vector();
        for (int i = 0; i < 4; i++) {
            Wheel tmpWheel = new Wheel();
            wheels.add(tmpWheel);
        }
    }
    ...
```

1.3.2 Relation between model and metamodel

The purpose of the following diagrams is to show how a metamodel describes a model, using the same example. The UML metamodel used in the following diagrams is a simplified version of the UML metamodel - it contains all the elements required to describe this UML model. These diagrams illustrate the relations between the model elements and the metamodel elements.

Figure 1.4 illustrates how the metamodel describes the classes in the model. Classes in the model (Person class, Vehicle class, Car class and Wheel) are instances of the Class class in the metamodel. The Class class in the metamodel do not describe the content of the classes such as operations and attributes.

Figure 1.5 illustrates how the metamodel describes the class attributes in the model. Class attributes in the model (name attribute of class Person and isDeflected attribute of class Wheel) are instances of the Attribute class in the metamodel.

Figure 1.6 illustrates how the metamodel describes the class methods (or operations) in the model. Class methods in the model (canMove() method of class Car) are instances of the Operation class in the metamodel.
Figure 1.7 illustrates how the metamodel describes datatypes used in the model: class attributes and return values in methods. Datatypes in the model are instances of the classes that extend the abstract `Datatype` class in the metamodel. In this example, `name` attribute (of class `Person`) type `String` instantiates class `String` in the metamodel, `isDeflated` attribute (of class `Wheel`) type `boolean` instantiates class `Boolean` in the metamodel, and `canMove()` operation returns a value of type `boolean` that instantiates class `Boolean` in the metamodel.
1.3. MOF metadata architecture

Figure 1.5. Relation between model and metamodel: attribute instances

Figure 1.8 illustrates how the metamodel describes the associations in the model. Associations in the model (composition car-wheel and association car-driver) are instances of the Association class in the metamodel. Association class in the metamodel does not describes association-ends and generalization in the model.

Figure 1.9 illustrates how the metamodel describes the association-ends
in the model. An association-end is defined by its name, type, multiplicity. The association-end name is optional, and the type is automatically set to the class related with this association-end. Association-ends in the model are instances of the `AssociationEnd` class in the metamodel (which contains all attributes to describe these features).

Figure 1.10 illustrates how the metamodel describes the generalizations
1.3. MOF metadata architecture

Figure 1.7. Relation between model and metamodel: datatype instances

in the model. A generalization is the relation between a class and its superclass. Generalizations in the model are instances of the generalizations association between Classifier class in the metamodel.

To summarise this illustration of the relation between models and metamodels, it is clear that the metamodel describes all model elements. It is important to say that however the metamodel provided in this example is used to define the structure of a particular model, this metamodel describes
Figure 1.8. Relation between model and metamodel: association instances
all models refering to the same structure (class, attributes, associations, etc).
1.3. MOF metadata architecture

1.3.3 Relation between metamodel and meta metamodel

The next series of diagrams show how the meta metamodel describes a metamodel, using the same example. In fact these diagrams become more interesting as they illustrate how MOF describes the UML metamodel. Both MOF meta metamodel and UML metamodel used in the following diagrams are a simplified version. MOF meta metamodel contains all the
Figure 1.10. Relation between model and metamodel: generalization instances elements required to describe the simplified UML metamodel. These diagrams illustrate the relations between the metamodel elements and the meta metamodel elements.

Figure 1.11 illustrates how the meta metamodel describes the classes in the metamodel. Classes in the metamodel (ModelElement class, Classifier class, Class class, Datatype class, Boolean class, String class, Operation class, Attribute class, AssociationEnd class and Association class) are
instances of the **Class** class in the meta metamodel. The **Class** class in the meta metamodel do not describe the content of the classes such as operations and attributes.

![Diagram](image)

**Figure 1.11. Relation between metamodel and meta metamodel: class instances**

Figure 1.12 illustrates how the meta metamodel describes the class attributes in the metamodel. Class attributes in the metamodel (**name** attribute...
of class ModelElement and visibility attribute of class Feature) are instances of the Attribute class in the meta metamodel.

Figure 1.12. Relation between metamodel and meta metamodel: attribute instances

Figure 1.13 illustrates how the meta metamodel describes datatypes used in the metamodel: class attributes and return values in methods. Datatypes in the metamodel are instances of the classes that extend the abstract Datatype
1.3. MOF metadata architecture

class in the meta metamodel. In this example, name attribute (of class ModelElement) type String instantiates class PrimitiveTypes in the metamodel and visibility attribute (of class Feature) type VisibilityKind instantiates class VisibilityKind in the metamodel.

![Diagram showing the relation between metamodel and meta metamodel: datatype instances](image-url)

*Figure 1.13. Relation between metamodel and meta metamodel: datatype instances*
Figure 1.14 illustrates how the meta metamodel describes the associations in the metamodel. Associations in the metamodel are instances of the Association class in the meta metamodel. Association class in the metamodel does not describes association-ends and generalization in the metamodel.
1.3. MOF metadata architecture

Figure 1.15 illustrates how the meta metamodel describes the association-ends in the metamodel. Association-ends in the metamodel are instances of the AssociationEnd class in the meta metamodel.

Figure 1.15. Relation between metamodel and meta metamodel: association-end instances

Figure 1.16 illustrates how the meta metamodel describes the generalizations in the metamodel. Generalizations in the metamodel are instances of
the `Generalizes` association between `GeneralizableElement` class in the meta metamodel.

![Diagram](image)

*Figure 1.16. Relation between metamodel and meta metamodel: generalization instances*

To summarise the relation between metamodels and meta metamodels, it is clear that the meta metamodel describes all metamodel elements. More-
1.3. MOF metadata architecture

over, these diagrams illustrate how the MOF metamodel describes the UML metamodel, despite the fact that reduced versions of the metamodels are used.

The last relation of the MOF architecture is the relation between the meta metamodels and meta metamodels, i.e., how MOF describes MOF. To illustrate this relation, similar illustrations can be used.
1.4 Transformation using MDA

The role of MOF and metamodels in MDA transformations can be illustrated in an abstract way. Figure 1.17 presents an overview of the MDA transformation between PIMs and PSMs, and the role of MOF and the metamodels. These transformations follow mapping techniques based on metamodels, described with UML or MOF.

![Figure 1.17. MOF and metamodels in MDA transformations](image)

All the P*M to P*M transformations have the same abstract structure. Figure 1.18 shows the abstract MDA transformation used in P*M to P*M transformation. The source model is an instance of a metamodel. The target model is an instance of a target metamodel. Source metamodel and Target metamodel are both instance of MOF meta metamodel and define the transformation metamodel (which is itself an instance of MOF). The transformation metamodel describes the transformation model that the transformation tool is using.

This abstract model of the MDA transformation is implemented by a transformation tool presented in section 3.2.4. This abstract transformation model will be implemented also when the MTV project extend to integrate model transformations.
1.4. Transformation using MDA

Figure 1.18. P*M to P*M Transformation
Chapter 2

Description of the project

This chapter describes the requirements for the project Metamodel Transformation for Verification (MTV): building the basis for a generic tool.

This project objective is to build a modular GUI for model browsing using standard techniques like those described by the MDA methodology and using the MOF as the meta description of the model.

Another goal of this project is to generate the metamodel description in XMI[MOF] \(^1\) format giving a UML representation of the metamodel in XMI[UML] \(^2\) format. This task will be done by interfacing the software with the tools available by netbeans.org project, in particular the UML2MOF tool.

In summary, the project can be described as to model and build a modular GUI for metamodelling purposes:

1. One of the modules available generates the MOF description of a given metamodel described using UML.

2. The other module provides the model browsing mechanism functionalities.

\(^1\)XMI[MOF] stands for MOF description of the metamodel in XMI format (see section 3.2.1)

\(^2\)XMI[UML] stands for UML description of the metamodel in XMI format (see section 3.2.1)
Considering the first point, the software will be able to provide as output a XMI[MOF] (section 3.2.1) file giving an XMI[UML] (section 3.2.1) file with the metamodel described as UML. Considering the second point, the models that will be used for browsing, are provided in data files XMI that, in order to be interpreted, need to have a description of what the information means. This description is provided by the metamodel also described in a file already defined in the previously generated file. This task includes generating Java interfaces for the metamodels using the Java Metadata Interface (JMI) that allows an implementation of a dynamic and platform independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata. Having generated the JMI interfaces, it should be possible to navigate inside of models and metamodels allowing its exploration in a graphical (tree based) fashion.

**Expected Results**

Following what was presented in the project description, the expected results of this project are to develop a generic modular GUI with two modules:

1. One module that will be able to generate the MOF description of a given metamodel described using UML.

2. A second module that will provide the functionality for model browsing using both information from a model repository and from a MOF model description.

The MTV application will be the basis of a much generic and featured application, including model transformation and Domain Specific Language (DSL) visualization. This must be taken into account to develop a modular and extensible application.
Chapter 3

Analysis of the problem

The Metamodel Transformation for Verification: building the basis for a generic tool project consists of a software development based on Model Driven Architecture (MDA) and metamodeling techniques. The MTV application will integrate several external libraries also called API in the framework. All these technologies are described in this chapter.

The MTV application is being built from scratch in order to deliver a generic tool for model and metamodel browsing. In the future, MTV application will be extended to provide model transformation and Domain Specific Language (DSL) visualization. This must be taken into account for this project development. A special emphasis is dedicated on following a development process that provides documentation for each phase of the project as well as a prototype for validation, includes tests in the process and builds an object oriented architecture that guarantees modularity and extension for the software.

3.1 Overview of project phases

The development process used for developing this application is a waterfall process. To reduce the high risk level in this development process, especially for new system development, the second iteration contains an implementation phase focused on a prototype development. Figure 3.1 illustrates the development process of the MTV software.

The first iteration of the process consists of a requirement phase and an
3.1. Overview of project phases

Figure 3.1. Waterfall process for MTV

The analysis comprises a research in the areas of MDA and MOF and a survey of the software tools.

During the second iteration, the analysis & design phase is focused on the Use Cases and Story Boards. Follows a short implementation phase is dedicated to the realisation of a first simple prototype.

The third iteration starts with the validation of the Use Cases, the Story Boards and the prototype. The analysis & design phase concerns the elaboration of the MTV architecture. The implementation phase, followed by a test phase delivers the beta version of the software.

During the fourth iteration, the analysis & design phase is focus on the thesis writing, and also on new tasks for the MTV application. The test phase provides a bug report, extremely important as this project will be extended.
3.2 Survey of software tools used in this area

This section is focused in all the tools used by the MTV application. This includes a short description of each one of the components and APIs used such as XMI, JMI, Meta Data Repository (MDR).

3.2.1 XMI XML Metadata Interchange

The first motivation to create XMI was to produce a standard way to be able to have UML models in XML and to provide modellers ability to move UML models between tools. The second motivation was to save any MOF-based metamodel in XML, and any model conforming to a metamodel.

XMI is a mapping of MOF to XML. The XMI standard defines rules for generating Document Type Definition (DTD) and XML schemas from MOF metamodels. Models are instances of metamodels, while metamodels are instances of the meta metamodel (MOF) and XMI provides an interchange format for models and metamodels. Figure 3.2 illustrates standard mechanisms for tools using XMI as a metadata interchange format.

![Figure 3.2. XMI: standard mechanism for tools](image)

When applying XMI to MOF, a concrete textual syntax for MOF is defined. This is also true for all other languages to which a metamodel (instance of MOF) is defined.
3.2. Survey of software tools used in this area

XMI[UML] & XMI[MOF] notation

Since MTV provides a standard interchange format for metamodels both described in UML or MOF, a distinction between these two must be guaranteed. Taking this into account, to refer to the content of XMI files describing metamodels, a notation is being purposed that is used throughout this document:

XMI[UML] stands for UML description of the metamodel in XMI format. This XMI[UML] (that is MOF compliant) is obtained by using an UML Case tool that supports standard XMI 1.2 export procedure. This file is called metamodel<languageName>_UML.xmi by the application.

XMI[MOF] stands for MOF description of the metamodel in XMI format. This file is called metamodel<languageName>_MOF.xmi by the application.

XMI files describing models do not have any specific notation, and are called <languageName><modelName>.xmi by the application.

3.2.2 JMI Java Metadata Interface

The Java Metadata Interface (JMI) is a set of interface to access metadata to handle the information provide by metamodels in order to do some operations such as code generation, model transformation and generic metadata browsing. This standard enables interoperability on the API level rather than on the XML level.

There are two types of interfaces for manipulating models, the JMI Reflective interfaces and the JMI Specific interfaces that will be described in the next lines.

JMI reflective interfaces

The JMI Reflective interfaces are not built to manipulate one type of model. They do not depend on a particular metamodel but on the meta metamodel. So they can be used on any type of model.

In addition, the JMI reflective interfaces provide the ability to move between meta levels. From any element of a model, using the methods provided by these interfaces, it is possible to explore its metaclass, in other words, the
class in the metamodel that describes this element, and finally, to retrieve
the metadata of this element.

The JMI reflective package is the set of interfaces that provides these
functionalities. This section describes each interface of the package. Figure
3.3 illustrates the JMI reflective package.

![JMI reflective interfaces](image)

**Figure 3.3. JMI reflective interfaces**

**RefBaseObject** The RefBaseObject interface is extended by all other re-
flexive interfaces. It provides common operation for testing object
identity, returning an object’s metaobject, and returning its facility
container as required for implementing structural constraints [4].
This interface is the key to move between meta levels. It provides the
refMetaObject() method which returns the metaobject that describes
the object in its metamodel specifications.

**RefPackage** The RefPackage interface is an abstraction for accessing a col-
clection of objects and their associations. The interface provides an
operation to access the metaobject description for the package, and
operations to access the package instance’s class proxy objects and its
association objects [4].
The RefPackage interface represents the package which contains all
model elements describing the metamodel. This interface can be used to manipulate metamodels, for instance to store a metamodel.

**RefAssociation** The RefAssociation interface provides the metaobject description of an association. It also provides generic operations querying and updating the links that belong to the association [4]. This interface is the key to retrieve the links between model elements such as aggregation, association and composition.

**RefFeatured** This interface inherits from RefBaseObject and contains all operations to get to the properties of an element in a model independent way [4].

**RefClass** The RefClass interface provides the metaobject description of a class proxy object, and a range of operations for accessing and updating an objects classifier scoped features [4]. This interface represents classes and provides operations to retrieve superclasses, attributes and methods.

**RefObject** The RefObject interface provides the metaobject description of an instance object, and a range of operations for accessing and updating the objects features [4].

**JMI Specific**

The JMI Specific interfaces are well adapted to manipulate models of one type. They offer specific operations that enable browsing in all models instance of a metamodel, since they reflect exactly the structure of a specific metamodel.

Specific JMI interfaces are all inheriting from the reflective interfaces. The inheritance patterns are [4]:

- An instance object that has no supertypes extends RefObject. All other instance objects extend their supertypes.

- A package object that has no supertypes extends RefPackage. All other instance objects extend their supertypes.

- All Class Proxy objects extend RefClass.
• All Association extend RefAssociation.

Figure 3.4 illustrates the relation between JMI reflective interfaces and JMI specific interfaces generated from a given metamodel (taken from NetBeans MDR example).

To summarise, it is evident that the reflective interfaces provide almost the same functionalities as the metamodel specific interfaces. In addition, with the JMI specific interfaces, an application would be able to navigate through models and metamodels. This property is an essential feature for the
MTV application, which can be considered as a generic metadata browser. The JMI specific interfaces are more convenient to use, and they will be widely used in the extension of the MTV project, e.g. model editing, model creation, and visual model exploration.

### 3.2.3 MDR Meta Data Repository

As the JMI interfaces provide the way to access metadata, the Meta Data Repository (MDR) provides the solution to store and manage metadata. The JMI interfaces are used to access the metadata in the metadata repository.

The Meta Data Repository (MDR) that is being developed as part of NetBeans open source project enables to load any MOF metamodel and store instances of that metamodel. MetaModels can be imported into the repository using XML metadata interchange format. This format is also used to export metadata from the repository. Metadata in the repository can be accessed using the JMI specific interfaces or the JMI reflective API.

#### MDR Architecture

MDR consists of several libraries developed as part of the NetBeans open source project. These libraries are independent from the NetBeans project and can be used by any tools. Figure 3.5 [5] illustrates the structure of MDR in the NetBeans project, including the JMI API.

The light-blue boxes represent the NetBeans project modules and the dark-blue boxes represent the libraries. Each module consists of one jar file and zero or more external libraries (grey boxes). Module dependencies are shown with the arrows.

The JMI Utilities module provides a JMI interfaces generator, a XMI importer and exporter. This module is an external library, independent from NetBeans project. The MDR engine is the standalone metadata repository, independent from NetBeans project. The MDR module is the NetBeans module that integrates this MDR engine with the NetBeans project. Then the MDR API module is an extension to JMI API for events and facility (?).
To summarise, this API is able to store and manage metamodels, generate the JMI specific interfaces for a metamodel and provide XMI facilities such as read and write. All these functionalities are required by the MTV application, so the MDR API must be integrated in the framework.
3.2. Survey of software tools used in this area

3.2.4 UML2MOF

The *NetBeans UML2MOF Tool* is a tool that converts an UML 1.4 description of a metamodel in XMI format into a MOF 1.4 description of a metamodel in XMI format.

The transformation from UML to MOF is defined by the *UML Profile for MOF*. This profile is based on OMG’s UML profile for MOF described in chapter 6 of the *UML Profile for EDOC* [6].

Figure 3.6 describes the metamodel transformation from UML to MOF using the *NetBeans UML2MOF Tool*. This is a concrete implementation of the MDA transformation described in the introduction (see 1.18).

![Figure 3.6. UML2MOF transformation](image)

In terms of implementation, the tool uses MDR to load the XMI[UML] file into the MDR repository using an XMI reader. Then it uses the JMI specific for MOF, JMI specific for UML (provided with the tool) and JMI reflective to transform the metamodel into MOF in the repository. Finally, it exports the metamodel into a XMI[MOF] file using a XMI writer.

It is important to highlight that the tool was tested with the XMI generated by two UML design tools, *Poseidon* 1.5 or higher, and *MagicDraw*.
Chapter 3. Analysis of the problem

6.0 [7]. This means that a user who wants to describe a metamodel in UML with a case tool in order to export it in XMI and convert it with the tool should use the templates files provided in the MDR website. However, there are some restrictions on the model design to succeed in the transformation process. A frequent transformation failure is caused by the fact that all associations must have a name in the model, and many people do not give a name to associations.

To summarise, it is clear that the NetBeans UML2MOF Tool provides an essential feature for the MTV application. The MDR repository is based on XMI[MOF] metamodels, and Case tools export metamodels in XMI[UML]. However, the use of this tool constraints the user to design metamodels in UML using case tools that support this standard. The UML Profile for MOF used by the NetBeans UML2MOF Tool adds some restriction to the design of the metamodel.

The NetBeans UML2MOF Tool is a concrete implementation of the MDA transformation and it would be particularly interesting to study the transformation algorithm. A potential extension of the MTV is to provide model transformation, and a similar process could be used.
Chapter 4

Use Cases

4.1 Primary and Secondary Actors

Primary Actors

<table>
<thead>
<tr>
<th>Primary Actors</th>
<th>Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>use the MTV to browse metamodels &amp; models</td>
</tr>
</tbody>
</table>

Secondary Actors

- JMI API
- NetBeans UML2MOF Tool
- NetBeans MDR API

4.2 Create a project in the MTV

Description

This Use Case describes the project creation in the MTV. This action occurs when the user selects the option to create a new project. The System creates a new folder with the proper structure for this new project (see 7.1.3). All the data related with this project will be stored in this location. The system uses external APIs to generate the mandatory files for model browsing. After the creation process, model browsing should be enabled.
Chapter 4. Use Cases

Definition

Name: Create a project in the MTV

Trigger: The user selects the option to create a new project

Actors: User (primary), NetBeans UML2MOF Tool (secondary), NetBeans MDR API (secondary), JMI API (secondary)

Assumptions:

1. The namespace of the UML diagram originally designed must be MetaModel <projectName>.
2. The metamodel Package in the UML diagram originally must be <projectName>.

Event Flow:

1. The User selects the option to create a new project.
2. The System asks the user to select a XMI[UML] file as metamodel for this project.
5. The System parses the file to extract project name.
6. The System creates the project structure.
7. The System uses NetBeans UML2MOF Tool to translate the XMI[UML] file into XMI[MOF].
8. The System saves XMI[UML] and XMI[MOF] files in the project folder.
9. The System loads the metamodel (XMI[MOF]) in the MDR repository.
10. The System uses NetBeans MDR API to generate the JMI specific interfaces for this metamodel.
11. The System compiles the JMI interfaces.
12. The System displays the metamodel project panel.

Extensions:
4.3. Open a project in the MTV

4.a The XMI[UML] file is malformed.
   1. The System writes an error message in the log window. The project is not created.
   2. back in 3.

5.a A project with the same name already exists.
   1. The System writes an error message in the log window. The project is not created.
   2. back in 3.

7.a NetBeans UML2MOF Tool generates an error.
   1. The System writes an error message in the log window. The project is created but metamodel, and model browsing is disabled.
   2. go to 12.

8.a The XMI[MOF] file is malformed.
   1. The System writes an error message in the log window. The project is created but metamodel and model browsing is disabled.
   2. go to 12.

10.a JMI generation failed.
   1. The System writes an error message in the log window. The project is created, metamodel and model browsing is enabled. Using the JMI for model creation is not possible.
   2. go to 12.

11.a JMI compilation failed.
   1. The System writes an error message in the log window. The project is created, metamodel and model browsing is enabled. Using the JMI for model creation is not possible.
   2. go to 12.

4.3 Open a project in the MTV

Description
This Use Case describes how a user can open an existing project in the MTV.
Definition

Name: Open a project in the MTV

Trigger: The User selects the option to open a project.

Actors: User (primary)

Assumptions:

1. The project structure exists in the Projects folder of the MTV
2. A metamodel exists in the MetaModels folder of this project

Event Flow:

1. The User selects the option to open a project.
2. The System displays the list of all projects available.
3. The User selects the project to open.
4. The System checks for project content.
5. The System displays the metamodel project panel.

Extensions:

3.a The User cancel the action.
   1. exit.

4.4 Update metamodel in the MTV

Description

This Use Case describes the situation when the user decides to update the metamodel for an existing project. The metamodel for this project has been generated during the project creation process (see 4.2).
4.4. Update metamodel in the MTV

Definition

Name: Update MetaModel in the MTV

Trigger: The user selects the option to update a metamodel in the MTV

Actors: User (primary), NetBeans UML2MOF Tool (secondary)

1. A project in the MTV is already open
2. A metamodel exists in the MetaModels folder

Event Flow:

1. The User selects the option to update a metamodel in the MetaModel Tool
2. The System presents a warning message pointing that this process will overwrite the existing metamodel.
3. The System asks the user to select the new XMI[UML] file with the metamodel for the project.
6. The System uses NetBeans UML2MOF Tool to translate the XMI[UML] file to XMI[MOF].
7. The System saves XMI[UML] and XMI[MOF] files in the project folder.
8. The System displays the metamodel project panel.

Extensions:

2.a The User cancel the action.
1. exit.

3.a The User cancel the action.
1. exit.

5.a The XMI[UML] file is not valid.
1. The System writes an error message in the log window.
2. back in 3.
6.a The *NetBeans UML2MOF Tool* returns an error.

1. The System writes an error message in the log window.
2. back in 3.

### 4.5 Update JMI in the MTV

**Description**

This Use Case describes how a user can update the interfaces to access the metamodel of current project.

**Definition**

**Name:** Update interfaces to access a metamodel in the MTV

**Trigger:** The User selects the option to update the Java Metadata Interfaces

**Actors:** User (primary), *NetBeans MDR API* (secondary)

**Assumptions:**

1. A project in the MTV is already open
2. A metamodel exists in the MetaModels folder

**Event Flow:**

1. The User selects the option to generate the JMI.
2. The System presents a warning message pointing that this process will overwrite the existing JMI.
3. The System uses *NetBeans MDR API* to generate the JMI for this metamodel.
4. The System compiles the JMI in the project folder.
5. The System displays the metamodel project panel.

**Extensions:**

2.a The User cancel the action.

1. exit.
4.6 Browse metamodel in the MTV

Description
This Use Case describes how a user can browse a metamodel. This situation occurs when the user select the option to browse the metamodel of the current project.

Definition

Name: Browse metamodel in the MTV

Trigger: The User selects the option to browse the metamodel.

Actors: User (primary), JMI API (secondary)

Assumptions:
1. A project in the MTV is already open
2. To the project a metamodel is available (XMI[UML] and XMI[MOF] files)

Event Flow:
1. The User selects the option to browse the metamodel.
2. The System retrieves the metamodel from the MDR repository.
3. The System explores the metamodel structure and builds a tree representation.
4. The System browses the tree in the metamodel panel.
4.7 Browse a model in the MTV

Description
This Use Case describes how a user can see the structure of a model. This situation occurs when the user selects the option to browse a model.

Definition
Name: Browse a model in the MTV

Trigger: The user selects the Browse Model option.

Actors: User (primary), JMI API (secondary)

Assumptions:
1. A project in the MTV is already open.
2. To the project a metamodel is available (XMI[UML] and XMI[MOF] files).
3. At least, one model is available in this project.

Event Flow:
1. The User selects the option to browse a model.
2. The System loads the model from the XMI file in the project folder into the MDR repository.
3. The System explores the model using the JMI reflective API.
4. The System builds a tree that represents the model structure.
5. The System browses the tree in the model panel.

4.8 Upload a model in the MTV

Description
This Use Case describes how the user can upload a model in the project folder using the MTV. This situation occurs when the project is already open, a metamodel is available in the project folder, and the user selects the option to upload a model in the project.
4.8. Upload a model in the MTV

**Definition**

**Name:** Upload a model in the MTV

**Trigger:** The User selects the option to upload a model.

**Actors:** User (primary)

**Event Flow:**

1. The User selects the option to upload a model.
2. The System asks the user to select the XMI file to load.
3. The User selects the XMI model.
4. The System copies the model in the project folder.
5. The System browses the model panel.

**Extensions:**

**3.a** A model with the same name already exists in the project.

1. The System display a warning message to the User.

3.a.1 • The user cancels the operation.

• back to 2.

3.a.2 • The user confirm the operation.

• go to 4.
Chapter 4. Use Cases

4.9 Diagram

The Use Case diagram 4.1 illustrates the interactions between the user and the System, as well as the interaction between the System and the secondary actors.

![Use Case Diagram of the MetaModelTool](image)

*Figure 4.1. Use Case Diagram of the MetaModelTool*
Chapter 5

Story Boards

This chapter presents the Story Boards for the Metamodel Transformation for Verification (MTV) application. A Story Board document contains pictures describing the behaviour of an application. These pictures are often made with design tools producing Graphical User Interface (GUI) without any functionality. The Story Boards document completes the Use Case document by providing an idea of what the software will look like. Therefore, the project can be validated by the customer.

Although sketches of MTV user interface were developed, since the MTV application has been already realised, the current version of the Story Board presents an illustration of the real MTV application behaviour rather than the prototype.

5.1 The MTV graphical user interface

This section presents the three main panels of the MTV, the welcome panel, the metamodel panel and the model panel.

Welcome Panel

The welcome panel is displayed when the applications starts (see Figure 5.1). When the user open or creates a project, this panel is hidden and the metamodel panel and model panel are visible.
Chapter 5. Story Boards

Figure 5.1. Welcome panel

The log area displays information to the user concerning the current process. For instance, when the user creates a project (see section 5.2), all information concerning the creation process is displayed. The button on the left of the log area provides the ability to clear it.

The menu bar is composed by the following menus:

- The File menu contains the options to open, create and close projects, as well as the exit option (see Figure 5.2).
- The MDR settings menu contains advanced options to browse information in the metadata repository (see Figure 5.3).
- The Help menu contains help options (see Figure 5.4).

The tool bar provides some of the functionalities available in the menu bar such as open/create/close a project options and the exit option.
5.1. The MTV graphical user interface

Figure 5.2. File menu

Figure 5.3. MDR settings menu

Metamodel panel

The *metamodel panel* presents all information on the project resources and provides a tool bar with browse and update options. Figure 5.5 shows the *metamodel panel*.

The green or red buttons in the panel inform the user on the project resources status. For instance, a green button for JMI sources and a red button for JMI classes means that the JMI specific interface for the metamodel were successfully generated but for any reason the compilation of the sources failed.

The first cluster of buttons in the tool bar offers the browsing options. The blue button with a ”‘J’” provides the JMI browsing option (see section 5.7), and the ”‘tree’” button provide the metamodel browsing option (see section 5.6). The orange button provides the XMI[UML] browsing and the red button the XMI[MOF] browsing.

The second cluster of buttons in the tool bar offers the update options. The first button with the blue arrow provides the metamodel update option (see section 5.4), and the second button with the wheels provides the JMI update option (see section 5.5).
Model panel

The *model panel* provide an interface to manage models (see Figure 5.6). The panel on the left contains the list of all models available for this project. The user can upload (see section 5.9) or delete a model from the project using the buttons on the tool bar. The tool bar provides also the option to browse a model (see section 5.8).
5.2 Create a project in the MTV

This story board describes the process of creating a project in the MTV. The user selects the option to create a new project in the tool bar (see Figure 5.7) or in the menu. Then the application displays a option pane to the user and asks him to select the XMI[UML] file as metamodel for the new project. The user selects the XMI[UML] file (see Figure 5.8), and the application extracts all information from this file to create a new project, transforms this file into XMI[MOF], copies both file in the project folder and generates the JMI specific. All information during this creation process is displayed on the log area. When the creation process is finished, the system displays the metamodel panel (see Figure 5.9).
5.3 Open a project in the MTV

This story board describes the process of opening an existing project in the MTV. The user selects the option to open a project in the tool bar (see Figure
5.3. Open a project in the MTV

Figure 5.9. The system displays the metamodel panel

5.10) or in the menu. The system retrieves all existing projects and shows an option pane asking the user to select a project to open. The user selects a project to open (see Figure 5.11) and the system displays the metamodel panel for this project (see Figure 5.12).

Figure 5.10. The user selects the open project option
Figure 5.11. The user selects the project to open
5.4. Update metamodel in the MTV

This story board describes the process of updating a metamodel in the MTV. A project is already open, and the user selects, in the metamodel panel tool bar, the option to update a metamodel (see Figure 5.13). The System presents a warning message pointing that this process will overwrite the existing metamodel (see Figure 5.14). The system displays an option pane asking the user to select a file. The user selects a XMI[UML] file (see Figure 5.15). The system converts the file into XMI[MOF] and copies both XMI[UML] and XMI[MOF] in the project folder. The system displays the metamodel panel (see Figure 5.16).
Figure 5.13. The user selects the update metamodel option

Figure 5.14. The system displays a warning message

5.5 Update JMI in the MTV

This story board describes the process of updating the JMI interfaces in the MTV. A project is already open and the user selects the option to update
5.5. Update JMI in the MTV

Figure 5.15. The user selects a XMI[UML] file

Figure 5.16. The system displays the metamodel panel

the JMI (see Figure 5.17). The System presents a warning message pointing that this process will overwrite the existing JMI (see Figure 5.18). If the user confirms the action, the system generates the JMI specific interfaces for the metamodel in the project, and compiles them. Finally, the system displays
the metamodel project panel (see Figure 5.19).

*Figure 5.17. The user selects the update JMI option*
5.6 Browse a metamodel in the MTV

This story board describes the process of browsing a metamodel in the MTV. The user selects the browse metamodel option in the metamodel panel tool.
bar (see Figure 5.20). The system explores the metadata and display a diagram oriented view of the metamodel (see Figure 5.21). This view is convenient to have a quick overview of all elements in the metamodel, but the user can modify the metamodel view (see Figure 5.22) to the package oriented view (see Figure 5.23), which offers a better view of the metamodel structure.

Figure 5.20. The user selects the browse metamodel option
5.7. Browse JMI in the MTV

This story board describes the process of browsing the JMI in the MTV. The user selects the browse JMI option in the *metamodel panel* tool bar.
Chapter 5. Story Boards

Figure 5.23. The system displays a package oriented view of the metamodel (see Figure 5.24). The system explores the JMI classes and display a “tree” representation of the JMI interfaces (see Figure 5.25). This view is convenient to have a quick overview of the interfaces generated for the metamodel and their methods.

Figure 5.24. The user selects the option to browse the JMI
5.8. Browse a model in the MTV

This story board describes the process of browsing a model in the MTV. The user selects the model he wants to display and then selects the browse model option in the model panel tool bar (see Figure 5.26). The system explores the model structure (using the JMI reflective interfaces to access the metamodel in the metadata repository) and displays a diagram oriented view of the model (see Figure 5.27).

*Figure 5.25. The system displays a tree representation of the JMI*
5.9 Upload a model in the MTV

This story board describes the process of uploading a model in the MTV.

The user selects upload model option in the model panel tool bar (see...
5.9. Upload a model in the MTV

Figure 5.28). The system displays an option pane asking the user to select a model (in XMI format). The user selects the model to upload (see Figure 5.29). The system checks the integrity of the file and copies the file in the project folder. The system updates the list of available models (see Figure 5.30).

![Figure 5.28. The user selects the option to upload a model](image)

*Figure 5.28. The user selects the option to upload a model*
Figure 5.29. The user selects the model to upload

Figure 5.30. The model has been uploaded in the project
Chapter 6

Architecture

This chapter presents the MTV software architecture and provides an overview of the MTV framework with a description of all its packages.

6.1 MTV Architecture

The architecture adopted for this software is a responsibility-based layering strategy [8]. The MTV software is layered based upon the following responsibilities:

- presentation logic
- domain model logic
- data access logic

The presentation layer is responsible for the presentation logic associated with an entity, such as the rendering of this entity in the user interface.

The domain model layer captures the most important types of objects in the context of the domain. The domain objects represent the entities that exist or events that occurs in the environment in which the system works [9].

The data access layer is responsible for making the state of an entity persistent.
This type of architecture is also called 3-layered architecture, or 3-tier architecture. It improves the development and maintenance of the software because responsibilities are isolated from one another. In addition, this layering strategy provides the advantage of modularity. The presentation layer could be a Graphical User Interface (GUI) using Java Swing components, or a website using Java Server Pages (JSP) and Servlets without any modifications on the two other layers.

Figure 6.1 illustrates the package dependency in the responsibility-based layering strategy.

![Diagram of package dependency in the responsibility-based layering strategy]

The layers described above are mapped into different packages. The presentation layer is mapped depending on the entity to manage with, such as models, metamodels and projects. For instance, the model package contains all classes responsible for model management, such as event listeners, buttons and display components.

The domain model layer is layered in four packages, based upon a reuse-based layering.
6.2 MTV packages overview

The purpose of this section is to provide an overview of MTV packages. Figure 6.2 illustrates the package dependencies in MTV architecture.

**project**

The `project` package is part of the domain model layer. This package manages all information relative to projects. This information is updated during runtime, and defines which functionalities are enabled or not. This package is also responsible for project folders creation.

The class `MetaModelProjectManager` (see 7.1.5) is the entry point to all functionalities of this package.
mdrControl

The mdrControl package is part of the domain model layer. This package is the core of the MTV application as it is used in most of the MTV functionalities, like project creation, metamodel update, or model browsing. All these functionalities require a metamodel, which is stored by MDR repository, and this repository must be accessed through mdrControl package.

The mdrControl package also provides functionalities for XMI reading and writing, JMI generation process using NetBeans MDR API for these tasks, and manages the UML to MOF transformation process, using the NetBeans UML2MOF Tool.

The mdrControl package contains a manager class called MDRController (see 7.2.6) which is the entry point for all functionalities provided by this package.

mdrBrowse

The mdrBrowse package is part of the domain model layer. It contains all classes responsible for model and metamodel browsing. The MTV uses a tree based representation to browse models and metamodels.

systemUtilities

The systemUtilities package is part of the domain model layer. It controls "low level" tasks such as copying, compiling, and a log controller.

gui

The gui package is the presentation layer. It uses the Java Swing API and the Java AWT API to provide a user friendly interface to manage metamodels projects, browse metamodels and models.

gui.metaModel

The gui.metaModel package is part of the presentation layer. Obviously, this package contains all classes that are involved in the metamodel panel of
6.2. MTV packages overview

the MTV.

Figure 6.3 shows the gui.metaModel package overview. The main class of the package is the MetaModelPanel. This main panel contains a panel responsible for showing the project information to the user, and a tool bar, providing functionalities to manage metamodels.

![Figure 6.3. gui.metaModel package overview](image)

gui.model

The gui.model package is part of the presentation layer. This package contains all classes responsible for the model management, such as event listeners, buttons and display components.

Figure 6.4 shows the gui.model package overview. The main class of the package is the ModelPanel. This main panel contains a tool bar, providing functionalities to manage models, such as uploading and browsing options.

gui.bars

The gui.bars package is part of the presentation layer. It contains all classes responsible for the menus and the tool bar in the main frame of the application. Therefore, it contains all classes responsible for project management, such as opening, creating or closing a project.

Figure 6.5 shows the gui.bars package overview.
gui.logPanel

The gui.logPanel package is part of the presentation layer. This package contains all classes responsible of the log panel of the MTV. This panel display the events that occur in the application.

Figure 6.6 shows the gui.logPanel package overview.
6.3 External APIs

Figure 6.7 illustrates the MTV framework including the external APIs.

6.3.1 JMI API

JMI Reflective package

The JMI Reflective package is part of the JMI API. It contains the JMI reflective interfaces. These abstract interfaces can be used for accessing the metadata without explicit knowledge of a metamodel.

The mdrBrowse package uses the JMI Reflective package to move between meta levels, from models to metamodels. location: javax.jmi.reflect

Java Archive (JAR) file: jmi.jar
JMI specific for MOF

The JMI specific interfaces for MOF 1.4 is part of the JMI API. The `mdrBrowse` package uses this package to access specifically MOF metadata, for model and metamodel browsing.
6.3. External APIs

location: javax.jmi.model
JAR file: mof.jar

6.3.2 NetBeans MDR API

MDR package
The MDR package is the metadata repository core. The MTV application will use this API to manage the metadata in the MDR repository.
location: org.netbeans.api.mdr
JAR file: mdrapi.jar

JMI Utilities package
The JMI Utilities package provides functionalities for JMI generation and XMI writing and reading.
location: org.netbeans.lib
JAR file: jmiutils.jar

MDR API package
The MDR API package provide functionalities for event handlers.
location: org.netbeans.lib
JAR file: org.netbeans.mdr

6.3.3 NetBeans UML2MOF Tool

UML2MOF package
The *UML2MOF* package contains the source files to convert XMI[UML] files into XMI[MOF] files.
location: org.netbeans.lib.jmi.uml2mof
JAR file: uml2mof.jar

JMI specific for UML 1.4 package
The JMI specific for UML 1.4 contains the JMI specific interfaces for UML used for the UML to MOF transformation.
location: org.omg.uml
JAR file: uml-1.4.jar
6.3.4 Ant API

The Ant API is required for the compilation process, in order to manage on-the-fly compilation.

location: org.apache.tools.ant
JAR file: ant.jar

6.3.5 JMI inheritences in MTV framework

Figure 6.8 illustrates the relations between the JMI interfaces in the MTV framework. The MTV application generates several JMI interfaces specific to metamodels. The NetBeans UML2MOF Tool contains also JMI specific interfaces for UML. Finally, the JMI API contains the JMI specific for MOF and the abstract JMI reflective interfaces.

![Figure 6.8. JMI inheritance in MTV framework](image-url)
Chapter 7

Implementation

7.1 Project management

In the MTV, a project is relative to a language, and this language is described by its metamodel. The application should be able to browse any model relative to this metamodel.

MTV should be able to cope with management of several metamodels and models during runtime. This means that the application has to have the knowledge about which metamodel must be used as metadata for the current model. For instance, the application cannot read an UML model using the XML metamodel.

The solution for this problem is to provide a support for project management. Therefore, the user has to open (or create) a project before being able to use any MTV’s (meta)model browsing functionality. The list of models the application displays in the model panel are related to this metamodel. Therefore, any model selected by the user is related to the metamodel.

In terms of implementation, a MTV project is an object, instance of the MetaModelProject class (see 7.1.5), which contains all information the application needs. The project management is controlled by the MetaModelProjectManager class (see 7.1.5) that is the entry point to access the project information.
7.1.1 Project Name

The name of the project is retrieved from the XMI[UML] file that the user selects when he chooses the option to create a new project. The project name is the namespace in the UML design of the metamodel. The application uses an XML parser, the `XMLDocumentFacilities` class (see 7.2.6) to extract the name from this XMI[UML] file and at the same time, check whether the file is well formed.

7.1.2 Getting information from existing projects

When the application is launched, it gets the information related to the existing projects in MTV’s internal folder structure. The application parses the projects folders and builds a project list. At the same time, it sets the information on its resources for each project. This information is also required during runtime in order to manage whether an option is enabled or not in the presentation layer. This is a very important aspect of the implementation taking into account that this way of implementing MTV increases user’s friendliness and usability of the software.

7.1.3 Creating the structure of a new Project

Each MTV project has the same folder structure. The project structure is created ”‘on the fly’” by the application when the user selects the option to create a new project. In principle, during this creation process, all files required by the application to enable model browsing are generated.

The following diagram illustrates how files are organised in the `Projects` folder.

```plaintext
\Projects
  \<projectName>
    \MetaModels
      metamodel<projectName>_UML.xmi
      metamodel<projectName>_MOF.xmi
    \Models
      *.xmi
    \JMI
    \src
```
7.1. Project management

The `MetaModels` folder contains all files describing the metamodel, the UML description of the metamodel and the MOF description of the metamodel in XMI format, `metamodel<projectName>_UML.xmi` and `metamodel<projectName>_MOF.xmi` respectively. These files are copied to the project folder by the `FileCopier` class if the `NetBeans UML2MOF Tool` succeeds in the UML to MOF transformation. This transformation occurs when the user creates a new project, or updates the metamodel of a project.

The `JMI\src` folder contains the JMI specific sources for the metamodel project and the `JMI\build` contains the JMI specific classes. Both JMI sources generation and compilation are managed by the application during run-time, when the user creates a new project or select the option to generate the JMI.

The management of the files in these folders is done by the application. The user is not supposed to directly copy or override them. The tool provides the necessary functionalities in the case that the user wants to update the content of these folders. The application parses the folders during run-time to prevent execution crashes due to unexpected files manipulations.

### 7.1.4 Browsing Project Settings and Resources

When the user opens a project (or when the project creation process ends), the application displays a panel with all information about the resources of the project. This panel gets the information about current project resources from the `MetaModelProject` object (see 7.1.5) related to the project.

The application also needs these resources to handle enable/disable options. For instance, the user is unable to browse models if the related metamodel is not stored in the metadata repository.
7.1.5 Classes involved in project management

MetaModelProject

The MetaModelProject is responsible for managing metamodel project settings and resources. It knows if files describing this metamodel are available (XMI[UML] and XMI[MOF]) and if the JMI sources (*.java files) or the JMI classes (*.class files) are generated.

It also provides the relative path to project files. This information is necessary when the application needs to compile or load files relative to the project.

The MetaModelProject class knows the number of instances of the metamodel in the MDR repository. This information is required by the MDRController (see 7.2.6 or 7.3.5) class when it browses a model.

The application should be able to show a list of all models available in a project. Therefore, the MetaModelProject class has a list of all models for this project.

MetaModelProjectManager

The class MetaModelProjectManager is responsible for managing all projects. It has a list of all available projects and knows the current project (the project
currently used by the application). When starting the application, this class
parses the project folder and rebuild the project list.
When a user creates a new project, the `MetaModelProjectManager` checks
whether a project with the same name already exists, then builds the project
structure and sets the new project as the current one.
This class is the entry point for project management. The MTV packages
have to deal with `MetaModelProjectManager` to get information on a par-
ticular project. All the projects modifications are notified to the
`MetaModelProjectManager`.

7.2 MDR repository management

Since MOF is the meta metamodel, it can be used as language for describ-
ing metamodels. MDR repository uses MOF language to create and store
metamodels in its repository. This assumes that XMI[MOF] files are used
as description of metamodels, and this is why the MTV requires an UML to
MOF transformation tool.

7.2.1 Using NetBeans UML2MOF Tool

After the analysis phase, it was expected to integrate *NetBeans UML2MOF
Tool* as a .jar file in order to perform the operations needed for the MTV.
Once the *NetBeans UML2MOF Tool* integration was performed the first
time, we observed that consequently to each use of the API, the MDR repos-
sitory was not accessible anymore by the MTV. While inspecting the source
code of the API, it was found that this API uses the same default MDR
repository as MTV. The *NetBeans UML2MOF Tool* shut this repository
down once the transformation process is finished.

The solution found to go around with this problem was to adapt the *Net-
Beans UML2MOF Tool* sources to prevent this conflict. The modifications
are the following:

- make a copy of the source file that causes this trouble (the source file
  is `org.netbeans.lib.jmi.uml2mof.Main` and the copy is
  `ch.unige.smv.mtv.mdrControl.UML2MOFCaller`)
- remove the repository shut down from this file
• add a return value for UML to MOF transformation

It is impossible to do an extension of the original class in the *NetBeans UML2MOF Tool* because all methods in this class are private. The best solution would be to integrate the functionalities provided by this class in the MTV project.

### 7.2.2 XMI[UML] to XMI[MOF] transformation

This transformation is performed when the user chooses to create a new project or when the user decides to update the metamodel for the current project. This algorithm describes the events in the *UML2MOFCaller* class and shows a concrete implementation of model transformation using MDR and the JMI.

**Algorithm:**

1. Presentation layer calls *MDRController* class to convert XMI[UML] into XMI[MOF];
2. *MDRController* class calls *UML2MOFCaller* class;
3. *UML2MOFCaller* class instantiates MOF metamodel to create UML metamodel;
4. *UML2MOFCaller* class loads the XMI[MOF] describing UML metamodel in the metamodel extent (using XMI reader);
5. *UML2MOFCaller* class instantiates MOF metamodel to create an extent for the metamodel target (extent is almost empty);
6. *UML2MOFCaller* class instantiates UML metamodel (using JMI specific for UML) to create an extent for the metamodel source;
7. *UML2MOFCaller* class loads the metamodel source in the metamodel source extent (using XMI reader);
8. *UML2MOFCaller* class calls *org.netbeans.lib.jmi.uml2mof.Transformer* class to execute the transformation, both metamodel source extent and metamodel target extent are given in parameter;
9. If the transformation succeeds
7.2. MDR repository management

(a) UML2MOFCaller class writes transformation output stream in metamodel<projectName>_MOF.xmi file in Projects/<projectName>/MetaModels folder;

(b) UML2MOFCaller class returns true;

(c) MDRControler class calls systemUtilities.FileCopier to copy XMI[UML] file as metamodel<projectName>_UML.xmi to Projects/<projectName>/MetaModels folder;

(d) MDRControler class calls project.MetaModelProjectManager to update current project status (XMI[UML] and XMI[MOF] metamodels are in the project folder);

(e) MDRControler class returns true to the presentation layer;

10. If transformation fails

(a) UML2MOFCaller class returns false (no file generated);

(b) MDRControler class returns false (no file copied);

7.2.3 Load a metamodel in MDR repository

This section describes the process of a loading a metamodel in MDR repository. To load and store a metamodel in the metadata repository the application first needs to create a new instance of MOF, that is almost empty (a unique package of type ModelPackage). To prevent multiple instantiations, the application tries first to get this metamodel instance from the repository, and only if the result is null, it creates a new extent. Listing 7.2.3 shows a part of the MDRControler class and illustrates the previous description.

Listing 7.1. MDRControler.java

```
// try to get metamodelExtent
ModelPackage extent = (ModelPackage) repository.getExtent(metaModelName);
// if the metamodelExtent is null, we have to create it
if (extent==null) {
    try {
        extent = (ModelPackage) repository.createExtent(metaModelName);
    } catch (CreationFailedException e) {
        e.printStackTrace();
    }
}
```

...
Chapter 7. Implementation

Once the metamodel package is created in MDR repository, the application uses a XMI reader (**javax.jmi.xmi.XMISaxReaderImpl**), to parse the XMI[MOF] file in the project folder and loads the metadata from this file to the metamodel package in the MDR repository.

### 7.2.4 Generate JMI sources and compile JMI

This section describes the event flow in the *mdrControl* package when the user selects the option to create a new project or the option to update the JMI interfaces for the current project.

The assumption made here is that the metamodel for current project is already loaded in the MDR repository (see 7.2.3).

**Algorithm:**

1. Presentation layer calls **MDRController** class to generate JMI interfaces;
2. **MDRController** class gets current project from **MetaModelProjectManager** class;
3. **MDRController** class gets the metamodel from the MDR repository;
4. **MDRController** class uses **JMIStream** class to create an output stream;
5. **MDRController** class uses **org.netbeans.api.mdr.JMIMapper** class to generate the JMI sources;
6. If JMI generation succeeds
   (a) **MDRController** class calls **MetaModelProjectManager** to update current project status (JMI sources are generated);
   (b) **MDRController** class calls **FileCompiler** to compile JMI sources during runtime;
   (c) **FileCompiler** returns compilation result to **MDRController**;
   (d) **MDRController** returns compilation result to presentation layer;
7. If JMI generation fails
   (a) **MDRController** returns false to presentation layer (no compilation);
7.2. MDR repository management

Figure 7.2. mdrControl package overview

7.2.5 Clean MDR repository

Several instances of metamodels might be created during run-time, as a model is an instance of a metamodel. Therefore, each time the user browses a model with the MTV, an instance of the metamodel is created in the MDR repository. All these instances are stored in a Btree database, a combination of a data file and an index file. Therefore, it might be a huge number of unused instances in the MDR repository. This means that MTV must remove all these metamodel instances when the application shuts down.

In terms of implementation, during the shut down process, the `cleanRepository()` method in class `MDRControler` is called. This method parses all packages in the repository and deletes them if they are not MOF instances. Metamodels stored in the MDR repository are instances of MOF, and will stay in the repository until the next execution of the program.

7.2.6 Classes involved in MDR management

**MDRControler**

This class is the entry point to MDR repository. It provides functionalities:

- create and get an extent in the repository
• load a metamodel (using XMILoader)

• load a model (using XMILoader)

• generate JMI interfaces (using org.netbeans.api.mdr.JMIMapper)

• clean MDR repository

• convert XMI[UML] to XMI[MOF] (using UML2MOFCaller)

• instantiate a metamodel

JMIStream

JMIStream class extends abstract class JMIStreamFactory. It is used by the JMIMapper class to obtain output streams for generating JMI interfaces. JMIStreamFactory Subclasses should implement the createStream(List, String, String) method to create/open an output stream based on a provided package name, class name and file extension.

UML2MOFCaller

UML2MOFCaller class is an adaptation of the main class in NetBeans UML2MOF Tool. This class is used by MDRController to convert XMI[UML] files into XMI[MOF] (see 7.2.2 for further information on this transformation process).

XMIExporter

XMIExporter class uses org.netbeans.api.xmi.XMIWriter class to export current model in XMI. This class is used by MDRController class when a user generates a model.

XMILoader

MDRController class uses XMILoader class to load models and metamodels from XMI files. XMILoader class uses org.netbeans.lib.jmi.xmi.XMISaxReaderImpl class to parse the XMI file and load the metamodel or model in the RefPackage given in parameter.
XMIFilter

XMIFilter class extends FileFilter class and checks if file's extension is XMI or XML. The application uses this class to filter the files that the user will select when he creates a new project, updates current project metamodel or uploads a model.

XMLDocumentFacilities

XMLDocumentFacilities class uses a XML parser to extract project name from the XMI[UML] file. The application uses this class during the project creation process.

7.3 MDR Browsing

Another important feature of this software is to browse the metadata stored in the MDR repository. The application uses the JMI reflective interfaces and the JMI specific interfaces for MOF to explore metamodels and models. This section presents the main algorithms to explore metadata in the MDR repository. In addition to the functionalities described in the Use Cases, some complementary browsing options are available providing essential information about the metamodel structure. A diagram oriented view and a package oriented view are available to represent metamodels, as well as a tree representation of the JMI specific interface.

In terms of implementation, the package responsible for these functionalities is the mdrBrowse package. This package communicates with the MDR repository through the org.netbeans.api.mdr.MDRManager, to retrieve the metadata.

7.3.1 Browse all metadata in the MDR repository

This function is used to browse all metadata stored in the MDR repository. This view offers a better comprehension on the MDR repository and how the metadata is stored (using MOF meta metamodel).

Algorithm:
Chapter 7. Implementation

1. **MDRViewer** object creates the tree root;

2. **MDRViewer** object calls `org.netbeans.api.mdr.MDRManager` object to get the list of all extents in the MDR repository;

3. for each name in the list:
   
   (a) **MDRViewer** object calls `org.netbeans.api.mdr.MDRManager` object to get the corresponding `RefPackage` package in the MDR repository;

   (b) if current package is an instance of `ModelPackage`, in other words, if this package contains a metamodel description then:

   i. **MDRViewer** object casts current package as `ModelPackage`;

   ii. **MDRViewer** adds current package name as a node in the tree;

   iii. **MDRViewer** object calls the `refAllClasses()` method to get the collection of all classes in MOF meta metamodel;

   iv. for each class in the collection:

      • **MDRViewer** object casts current class as `RefClass`;

      • **MDRViewer** adds current class name as a node in the tree;

      • **MDRViewer** object calls the `refAllOfClass()` method to get the collection of all elements in the metamodel that are instance of the current class;

      • for each element in the current class:

         A. **MDRViewer** object casts current element as `ModelElement`;

         B. **MDRViewer** object adds current element name as a node in the tree;


7.3.2 Browse MetaModel with a diagram oriented view

This section describes the algorithm used by the application to explore the metamodel structure and build a diagram oriented tree that represents this metadata. This diagram oriented tree is customized with UML style icons in the presentation layer in order to provide a diagram oriented view. The diagram oriented view of the metamodel is convenient to have a quick overview of all elements in the metamodel.
Algorithm:

1. The presentation layer calls MDRViewer class to get a diagram oriented tree that represents the metamodel;

2. MDRViewer object calls org.netbeans.api.mdr.MDRManager object to get current metamodel project RefPackage package in the MDR repository;

3. MDRViewer checks if current package is not null and is an instance of MOF;

4. MDRViewer object casts this package as ModelPackage;

5. MDRViewer creates the tree root with package name;

6. MDRViewer object calls the refAllClasses() method to get the collection of all classes in MOF meta metamodel;

7. for each class in the collection:
   • MDRViewer object casts current class as RefClass;
   • if this class is an instance of MofClassClass class, this class is a class or a generalization in the metamodel, then:
     (a) MDRViewer object calls the refAllOfClass() method to get the collection of all elements in the metamodel that are instance of MofClassClass.
     (b) for each element in this collection:
       i. MDRViewer object casts current element as MofClass;
       ii. MDRViewer object adds current element name as a Class node in the tree;
       iii. MDRViewer uses the getSupertypes() method to get the collection of all super classes of current class;
       iv. for each super class in the collection:
         – MDRViewer object casts current element as MofClass;
         – MDRViewer object adds a Generalization node in the tree;
   • if current class is an instance of AssociationClass class, this class is an association in the metamodel, then:
     (a) MDRViewer object calls the refAllOfClass() method to get the collection of all elements in the metamodel that are instance of AssociationClass.
(b) for each element in this collection:
   i. **MDRViewer** object casts current element as **ModelElement**;
   ii. **MDRViewer** object adds an **Association** node in the tree;

### 7.3.3 Browse JMI

This section describes the algorithm used by the application to explore and browse a tree representation of the JMI specific interfaces generated for a given metamodel. The JMI view can be used to have an overview of the specific interfaces and its available methods.

This method uses the JMI reflective interfaces to explore the metamodel, and the `java.lang.reflect` package to access to the compiled JMI specific interfaces.

Algorithm:

1. The presentation layer calls **MDRViewer** class to get a tree that represents the JMI interfaces generated;
2. **MDRViewer** object calls **org.netbeans.api.mdr.MDRManager** object to get current metamodel project **RefPackage** package in the MDR repository;
3. **MDRViewer** object checks if current package is not null and is an instance of MOF;
4. **MDRViewer** object creates the tree root with package name;
5. **MDRViewer** object uses the **getClass()** to get the runtime class of this package;
6. **MDRViewer** object uses the **getDeclaredMethods()** to get all methods from this interface;
7. **MDRViewer** adds all these methods as **Method** nodes to the tree;
8. **MDRViewer** object calls the **refAllClasses()** method to get the collection of all classes in the metamodel;
9. for each class in the collection:
   (a) **MDRViewer** object casts current element as **RefClass**;
### 7.3.4 Browse MetaModel with a package oriented view

The *package oriented* view uses mainly the same algorithm as the *diagram oriented* view algorithm. The only difference is that this representation of the metamodel uses the `TestMOFClasses` class to build a view that offers a better representation of the metamodel structure.

### 7.3.5 Browse Model

The browse model algorithm also uses the same general algorithm to explore the metadata. In this particular case, the application must have access to the metaobject in order to retrieve attribute values. Listing 7.3.5 shows a
sample of code that illustrates some of the methods used to access to the
metaobject.

Listing 7.2. MDRModelBrowser.java

```
public void readAttributes(RefObject object, MetaModelNode nodeClass){
    // metaobject of any RefObject is an instance of MofClass
    MofClass metaObject = (MofClass) object.refMetaObject();
    // explore the object's metamodel to get object's attributes
    Iterator it = metaObject.getContents().iterator();
    while (it.hasNext()){  
        ModelElement element = (ModelElement) it.next();
        if (element instanceof javax.jmi.model.Attribute){
            // add current Element as an attribute of current class node
            MetaModelNode nodeAttribute = new MetaModelNode(
                element.getName() + " : " + object.refGetValue(
                    element).toString(), "Attribute");
            nodeClass.addChild(nodeAttribute);
        }
    }
}
```

7.3.6 Classes involved in MDR browsing

**MetaModelNode**

The **MetaModelNode** class has the responsibility to build a tree made of
**MetaModelNode** objects. The first attribute of this class is the children list
(**MetaModelNode** objects), the second attribute is the name, and the last is
the type (package, class, attribute, generalization, ...).

**MetaModelNodeComparator**

The **MetaModelNodeComparator** class implements the interface **Comparator**
to compare two **MetaModelNode** objects, using their names alphabetically
sorted. The comparator is used by the **MetaModelNode** class to sort the
children list.

**MDRViewer**

The **MDRViewer** class contains all methods for metamodel browsing. The
**getPackageFullInfoMetaModelTree()** method is described in 7.3.1, the
getJMITree() method is described in 7.3.3, the getDiagramOrientedMetaModelTree() is described in 7.3.2, the getPackageOrientedMetaModelTree() method is described in 7.3.4 and use the TestMOFClasses.

TestMOFClasses

The TestMOFClasses class is used when the application builds a package oriented view of a metamodel. To build the package oriented tree, the application needs to create a temporary structure that will be managed and stored in this class.

MDRModelBrowser

The MDRModelBrowser class contains all methods for model browsing.

7.4 Utilities

This section presents some of the functionalities provide by the systemUtilities package.

7.4.1 On the fly compilation

Another important feature of the MTV application is, after a project creation or metamodel update, the ability to compile the JMI specific interfaces for the specific metamodel without the need of restarting the application. MTV uses Ant API to provide "on the fly" compilation functionalities for the JMI interfaces. Listing 7.4.1 illustrates how to create a Ant project directly in the code, and compile the project as well.

```
Listing 7.3. FileCompiler.java

    public boolean compileJMIwithAntFromCode(String inputPath, String outputPath ){
        try{
            // create new Javac task
            Javac antTask = new Javac();
            Project antProj = new Project();

            // project initialisation
            antProj.init();
        }
```
// set source and destination directories
antTask.setProject(antProj);
antTask.setSrcdir(new Path(null, inputPath));
antTask.setDestdir(new File(outputPath));

// set java class path
antTask.setClasspath(new Path(null, System.getProperty("java.class.path")));

// execute ant task
antTask.execute();

} catch (BuildException exeption) {
    LogControler.getUniqueInstance().setState("File Compiler Error: " + exeption);
    return false;
}

LogControler.getUniqueInstance().setState("File Compiler: compilation succeeds");
return true;
This chapter provides a case study using the Concurrent Object Oriented Petri Nets (COOPN) metamodel and the *drink vending machine* model.

### 8.1 COOPN Metamodel

Figure 8.1 shows a simplified version of the COOPN metamodel.

![Figure 8.1. reduced and simplified COOPN metamodel](image)

Listing 8.1 shows a small part of the content of the XMI[UML] file generated by a Case tool.

1. The `<UML:Model>` node is the container for the metamodel, and the `name`
attribute must contain the substring `<languageName>` to be accepted by the MTV application. The `<UML:Package>` node is the metamodel package. It contains all the model elements. The `name` attribute must be `<languageName>` to be accepted by the application.

**Listing 8.1. metamodelCOOPNv4 UML.xmi**

```xml
<?xml version='1.0' encoding='UTF-8'?>
<XML xmlns:xmi='http://www.omg.org.xmi'>
<XML:header>
<XML:documentation>
<XML:exporter>Netbeans XMI Writer</XML:exporter>
</XML:documentation>
</XML.header>
<XML:content>
<XML:Model xmi.id='I596d42dm1043dde64ecmm7d6c' name='COOPNv4 MetaModel'
  isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
  <XML:Namespace ownedElement>
    <XML:Package xmi.id='I596d42dm1043dde64ecmm7d76' name='COOPNv4'
      isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
      <XML:Stereotype xmi.idref='I596d42dm1043dde64ecmm7d78'/>
    </XML:Package>
    <XML:ModelElement taggedValue>
      <XML:TaggedValue xmi.id='I596d42dm1043dde64ecmm7dd2'
        isSpecification='false'>
        <XML:TaggedValue.dataValue>org.netbeans.jmi</XML:TaggedValue.dataValue>
      </XML:TaggedValue>
    </XML:ModelElement.taggedValue>
    <XML:ModelElement.stereotype>
      <XML:Stereotype xmi.idref='I596d42dm1043dde64ecmm7d78'/>
  </XML:Namespace.ownedElement>
</XML:Model>
<XML:Stereotype xmi.id='I596d42dm1043dde64ecmm7d78' name='metamodel'
  isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'>
</XML:Stereotype>
<XML:Class xmi.id='I596d42dm1043dde64ecmm7c8f' name='Module'
  visibility='public'
  isSpecification='false' isRoot='false' isLeaf='false' isAbstract='false'
  isActive='false'>
  <XML:Classifier.feature>
    <XML:Attribute xmi.id='I596d42dm1043dde64ecmm7c7a' name='name'
      visibility='private'
```
8.1. COOPN Metamodel

Listing 8.1 shows the XMI[MOF] file generated with the *NetBeans UML2MOF Tool*.

Listing 8.2. *metamodelCOOPNv4 MOF.xmi*

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<%@xml version='1.2' xmlns:Model='org.omg.xmi.namespace.Model' timestamp='Mon Sep 05 16:18:13 CEST 2005'%>
</xml.header>
<xml.documentation>
<xml.exporter>Netbeans XMI Writer</xml.exporter>
<xml.exporterVersion>1.0</xml.exporterVersion>
</xml.documentation>
</xml.header>
<xml.content>
<model:Package xmi.id='a1' name='PrimitiveTypes' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
  <model:Namespace contents>
    <model:PrimitiveType xmi.id='a2' name='Integer' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
      <model:PrimitiveType xmi.id='a3' name='Long' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
        <model:PrimitiveType xmi.id='a4' name='Float' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
          <model:PrimitiveType xmi.id='a5' name='Double' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
            <model:PrimitiveType xmi.id='a6' name='Boolean' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
              <model:PrimitiveType xmi.id='a7' name='String' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
                <model:Namespace contents>
                  <model:Package xmi.id='a8' name='COOPNv4' annotation='' isRoot='true' isLeaf='false' isAbstract='true' visibility='public_vis'>
                    <model:Namespace contents>
                      <model:Tag xmi.id='a9' name='javax.jmi.packagePrefix' annotation='' tagId='javax.jmi.packagePrefix'>
                        <model:Tag.values>org.netbeans.jmi</model:Tag.values>
                      </model:Tag>
                    </model:Namespace contents>
                  </model:Package>
                </model:Namespace contents>
              </model:PrimitiveType>
            </model:PrimitiveType>
          </model:PrimitiveType>
        </model:PrimitiveType>
      </model:PrimitiveType>
    </model:Namespace contents>
  </model:PrimitiveType>
</model:Namespace contents>
</model:Package>
</xml.content>
</xml>
Chapter 8. Case Study

<Model:Tag.elements>
  <Model:Package xmi.idref='a8'/>
</Model:Tag.elements>
</Model:Tag>
</Model:Association>
<Model:Association xmi.id='a10' name='contextinterface_contains_method'
  annotation='' isRoot='false' isLeaf='false' isAbstract='false'
  visibility='public_vis' isDerived='false'>
</Model:Namespace.contents>
<Model:AssociationEnd xmi.id='a11' name='contextinterface_from_method'
  annotation='' isNavigable='true' aggregation='shared'
  isChangeable='true'>
  <XMI.field>0</XMI.field>
</Model:AssociationEnd>
</Model:Association>
<Model:Class xmi.idref='a12'/>
</Model:Namespace.contents>
<Model:AssociationEnd xmi.id='a13' name='method' annotation='' aggregation='none'
  isChangeable='true'>
  <XMI.field>0</XMI.field>
</Model:AssociationEnd>
</Model:Namespace.contents>
<Model:Class xmi.idref='a14'/>
</Model:Namespace.contents>
<Model:Attribute xmi.id='a15' name='methodName' annotation='' scope='instance_level'
  visibility='public_vis' isChangeable='true' isDerived='false'>
  <XMI.field>1</XMI.field>
</Model:Attribute>
</Model:NamespaceCONTENTS>
</Model:Class>
</Model:Extension>
Listin 8.1 shows one of the JMI specific interface generated, the ADT class in the COOPN metamodel.

Listing 8.3. ADT.java

```java
package org.netbeans.jmi.coopnv4;

/**
 * ADT object instance interface.
 */
public interface Adt extends org.netbeans.jmi.coopnv4.Module {

    /**
     * Returns the value of reference adtbody.
     * @return Value of reference adtbody.
     */
    public org.netbeans.jmi.coopnv4.Adtbody getAdtbody();

    /**
     * Sets the value of reference adtbody. See \{@link #getAdtbody\} for description
     * on the reference.
     * @param newValue New value to be set.
     */
    public void setAdtbody(org.netbeans.jmi.coopnv4.Adtbody newValue);

    /**
     * Returns the value of reference adtinterface.
     * @return Value of reference adtinterface.
     */
    public org.netbeans.jmi.coopnv4.Adtinterface getAdtinterface();

    /**
     * Sets the value of reference adtinterface. See \{@link #getAdtinterface \} for description on the reference.
     * @param newValue New value to be set.
     */
}
```
Figure 8.2 shows a *package oriented* view of COOPN metamodel. The root of the tree is the COOPNv4 package. It contains all the metamodel elements (classes, associations, etc).

The next level of the tree shows the classes of the metamodel: ADT class, ADTBody class, ADTInterface class, ClassBody class, ClassInterface class, ...

This picture focused on a particular association related to the ADT class: the \texttt{adt\_has\_body} association is composed by two association-ends, \texttt{adt\_from\_body} of type \texttt{ADTBody} and \texttt{adtbody} of type \texttt{ADT}.

The ADT extends Module generalization describes that the ADT class extends the Module class in the metamodel.
Figure 8.2. Package oriented view of the COOPN metamodel
8.2 COOPN drink vending machine model

Listing 8.2 shows the COOPN model in XMI extract from the drink vending machine model in COOPN. This extraction was done using the COOPN Source Extractor API together with the COOPN to COOPN XMI API using the JMI specific interfaces for COOPN.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<XMI xmi.version='1.2' timestamp='Wed Jul 27 15:05:04 CEST 2005'>
  <XMI.header>
    <XML.documentation>
      <XML.exporter>Netbeans XMI Writer</XML.exporter>
      <XML.exporterVersion>1.0</XML.exporterVersion>
    </XML.documentation>
  </XMI.header>
  <XML.content>
    <COOPNv4.Module xmi.id='a1' name='Money'/>
    <COOPNv4.Module xmi.id='a2' name='Coins'/>
    <COOPNv4.Module xmi.id='a3' name='Drink'/>
    <COOPNv4.ContextInterface xmi.id='a4'>
      <COOPNv4.Method xmi.idref='a5'/>
      <COOPNv4.Method xmi.idref='a6'/>
      <COOPNv4.Method xmi.idref='a7'/>
      <COOPNv4.Method xmi.idref='a8'/>
      <COOPNv4.Method xmi.idref='a9'/>
    </COOPNv4.ContextInterface.method>
    <COOPNv4.ContextInterface.context_from_interface>
      <COOPNv4.ContextInterface.context_from_interface>
      <COOPNv4.ContextInterface.use>
        <COOPNv4.Use xmi.idref='a11'/>
        <COOPNv4.Use xmi.idref='a12'/>
        <COOPNv4.Use xmi.idref='a13'/>
      </COOPNv4.ContextInterface.use>
      <COOPNv4.ContextInterface.gate>
        <COOPNv4.Gate xmi.idref='a14'/>
        <COOPNv4.Gate xmi.idref='a15'/>
        <COOPNv4.Gate xmi.idref='a16'/>
        <COOPNv4.Gate xmi.idref='a17'/>
      </COOPNv4.ContextInterface.gate>
    </COOPNv4.ContextInterface.context_from_interface>
  </COOPNv4.ContextInterface>
  ...
</XMI.xmi>
```

Figure 8.3 presents a diagram oriented view of the drink vending machine. The first level of the tree shows all elements in the model: three ADT objects, two Context objects, two ContextInterface objects, five CoopnClass objects, etc.

The next level of the tree provides the attributes of these objects and their value. For instance, ADT object has a name attribute of value Drink.
8.2. COOPN drink vending machine model

Figure 8.3. diagram oriented view of the drink vending machine
Chapter 9

Conclusion

The Metamodel Transformation for Verification (MTV) application provides the functionalities that were expected and documented at the beginning of the project. The MTV is a generic metadata browser able to browse in a tree based representation, models that refer to a MOF based metamodels. An UML to MOF transformation module is part of the MTV framework and can convert any UML description of a metamodel in XMI into a MOF description of a metamodel.

In addition to these functionalities, the MTV application can browse the metamodels themselves, in a diagram oriented or a package oriented view. The generated JMI specific interfaces can also be displayed in tree based representation, as well as the whole content of the MDR repository.

The MTV application provides a support for project management to cope with several metamodels and models during runtime and also suitable for organization purposes. All files related to a project, such as models, metamodels and JMI specific interfaces are stored in a specific folder. The application manages this folder and provides all options to the user to access to the content. Therefore, the user can update a metamodel, generate the JMI and upload models in the project.

The MTV 1.0 software is running as a generic metadata browser and encompasses the functionalities described above. However, some additional functionality should be integrated to this software, such as the option to clear all the metadata in the MDR repository and rebuild the metamodels.
from the XMI[MOF] files. Other options like the possibility to browse XML files in a tree based representation should be introduced. Finally, it would be useful to create an UML project using the existing JMI specific interfaces for UML, in order to browse any UML model in XMI format.

In what concerns the major extensions of the MTV software, a tool will be integrated in order to generate graphical representations instead of tree representations of metamodels and models. Another major extension would be to add transformations based on metamodels, and finally, to upgrade the MTV software to a general model editor, able to create, modify and browse any model that refer to a MOF based metamodel.

In summary of this bachelor project, I would say that it was particularly interesting for me to explore the world of metamodeling techniques, and realise how powerful it could be. It was challenging to understand the tools related to these technologies such as XMI, JMI and MDR, and integrates them into the MTV framework. But the most difficult task in this project, and the key for its realisation, was to find out how to use exactly the JMI to explore the metadata for metamodel and model browsing.

In terms of project realisation, it was the first time I had to manage a "real" application, with project management, APIs integration, friendly user interface, from the beginning (specifications) to its realisation: MTV 1.0.
Appendix A

Acronyms

API  Application Program Interface
CIM  Computation Independent Model
COOPN Concurrent Object Oriented Petri Nets
CWM  Common Warehouse Metamodel
DTD  Document Type Definition
DSL  Domain Specific Language
EJB  Enterprise Java Bean
IDE  Integrated Development Environment
ISO  International Organization for Standardization
JAR  Java Archive
JMI  Java Metadata Interface
JSP  Java Server Pages
GUI  Graphical User Interface
MOF  Meta Object Facility
MDA  Model Driven Architecture
MDR  Meta Data Repository
MTV  Metamodel Transformation for Verification
OCL  Object Constraint Language
OMG  Object Management Group
PIM  Platform Independent Model
PSM  Platform Specific Model
SMV  Software Modeling and Verification
UML  Unified Modeling Language
UC   Use Case
URL  Uniform Resource Locator
WWW  World Wide Web
XMI  XML Metadata Interchange
XML  Extended Markup Language
Bibliography


