

# A Critical Analysis of MDA Standards through an Implementation: the *ModFact* Tool

Xavier Blanc<sup>1</sup>, Salim Bouzitouna<sup>1</sup> and Marie-Pierre Gervais<sup>1,2</sup>

<sup>1</sup>Laboratoire d'Informatique de Paris 6 (LIP6)

<sup>2</sup>University Paris X

LIP6 - 8 rue du Capitaine Scott - F75015 PARIS

{Xavier.Blanc, Salim.Bouzitouna, Marie-Pierre.Gervais}@lip6.fr

## Abstract

*We present in this paper our experiment of implementing various MDA standards in a tool named ModFact. This implements MOF and XMI standards from OMG as well as JMI from Sun. Lessons learned from this experiment are described. In particular, we stress the limits we encountered when implementing the MOF standard. This leads us to wonder whether the MOF specification tackles the right level of standardization required in models manipulation in order to be really useful for software engineers adopting an MDA approach for their development.*

## 1. Introduction

The Model Engineering gains in importance especially since OMG issued the Model-Driven Architecture (MDA) initiative aiming at a global approach for integration and interoperability [1][2]. The MDA approach promotes the concept of model, i.e., advocates the rise in abstraction in the software development. According to the well-known principle of separation of concerns, the MDA advocates the isolation of business concerns from their technical achievement. The idea is that the business concerns can be modelled independently from any middleware concerns. Therefore, business models are not corrupted by technical concerns. This is strongly anticipated as a way of simplifying the construction of applications. In this way, the main part of the development becomes an activity upstream, dedicated to business concerns through the elaboration of the application model that abstracts away technical details, i.e., the so-called Platform-Independent Model (PIM). The transformation of a PIM into a Platform-Specific Model (PSM) is then achieved when introducing into the PIM the technical considerations depending on the chosen middleware.

One of the challenges that MDA initiative must face in order to be really efficient is the availability of tools supporting the activities of models engineering. Among them, we identify two fundamental functionalities that have to be supported: the models storage and the models transformation. Actually, the model transformation strongly depends on the model storage. For example, at least, it requires that the source model be stored in an electronic form to be processed by a transformation engine. It is then needed to define how models are represented, how the links between them are represented and how they are accessed. To face these issues, OMG has defined the MOF and XMI standards [4][5] and Sun has defined the JMI standard [6]. Consequently, models can be represented either by objects or by XML files. In each case, an API is available to manipulate models (e.g., editing, modifying or transforming models). Thus, thanks to these standards, models manipulation could be thought as a solved issue.

However, implementing these standards raises new problems. These ought to be tackled by these standards. Actually, we did such an implementation, which we provide as an Open Source tool named *ModFact* that is dedicated to models manipulation. *ModFact* enables the storage of models and the models transformation. We especially focus this paper on the models storage service. To offer it, *ModFact* implements the MOF, XMI and JMI standards.

This experiment enables us to identify drawbacks in the current standards, especially in the MOF standard. From our viewpoint, the part of this standard devoted to the repositories construction and the provision of an API enabling the browsing of models is not adapted for a large use in software development according to a model-driven approach. Actually, it seems to us that it does not tackle the right level of models manipulation required by software engineers involved in such a process. To explain

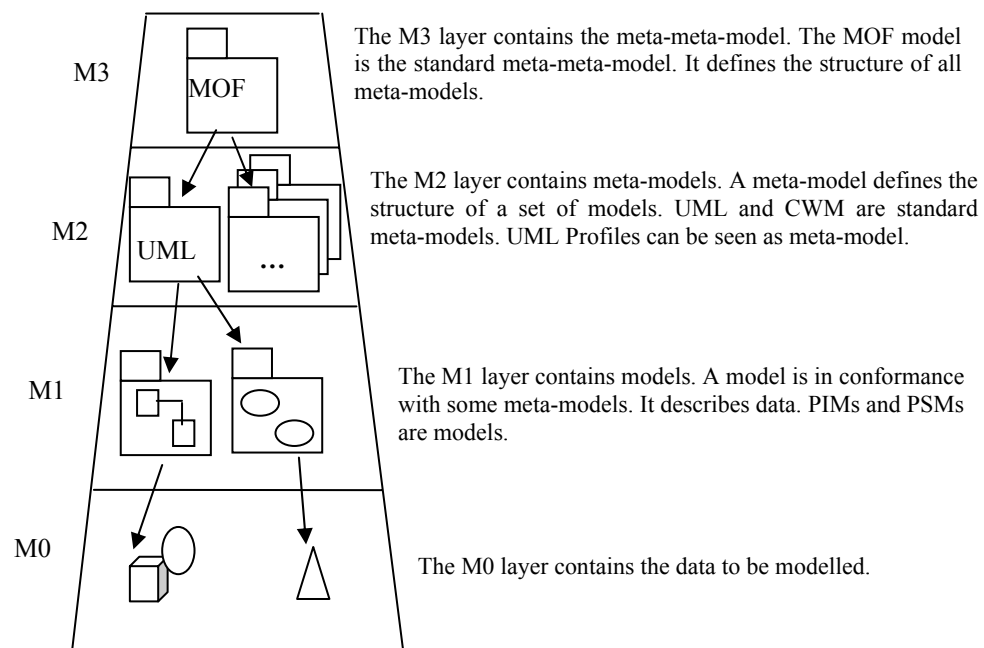
this, we compare the models storage functionality and the engineers requirements in terms of API with the data storage performed by DataBase Management Systems.

This paper is structured as follows. In Section 2, we provide the reader with a brief overview of the MDA standards we have considered in our experiment. We then present *ModFact* in Section 3. In addition to the tool description, we stress on the limits we encountered and how we overcame them. Section 4 relates our feedback on this experiment and our analysis of the MOF standard. The conclusion expresses the future developments of the tool and draws some issues to this work.

## 2. The MDA Standards

The MDA initiative comes with several standards. We introduce some of them that are relevant in our context. Let us notice that these standards are based on a common foundation that is the four layers architecture defined by the OMG, briefly introduced in Figure 1.

The MOF (Meta-Object Facility) standard deals with meta-modeling [4]. It defines the so-called “MOF model”, which is a meta-meta-model. The MOF model defines the structure of all meta-models. A meta-model defines some concepts and their relationships. The MOF model defines that a meta-model is a package in which each class represents a concept. Classes can be associated to express relationship between concepts. The MOF standard also defines a mapping of MOF meta-models (i.e., meta-models in conformance with the MOF model) onto CORBA IDL. This mapping is used to build API of models repositories, this API being defined as IDL interfaces. Thanks to such an API, a software engineer can create models, which are compliant with the meta-model used to build the models repository. He/she can use the API to browse through these models. Let us notice that even if IDL is used, no implementation is prescribed in the MOF standard for the models repositories. These can be a database, a file system or CORBA objects.



**Figure 1: The Four Layers Architecture**

The XMI (XML Model Interchange) standard is frequently referred as the standard for exchanging UML Models [5]. In fact, XMI is more powerful. It defines rules for building DTD (Data Type Definition) or XML Schema from any meta-model. Once the DTD (or XML Schema) is available, a software engineer can create models as XML files. So XMI can be used to exchange all kinds of models, not restricted to UML models.

The JMI (Java Metadata Interface) standard, elaborated by Sun, deals with the MDA as it defines a mapping of MOF meta-models onto Java [6]. Thanks to the JMI standard, it is possible to generate a Java API of a models repository from a MOF meta-model. As well as the IDL API, this Java API enables a software engineer to create a model and then to browse through it. The only difference is that this API is

full Java whereas the IDL API can be implemented in any language, provided that a projection from IDL to this language exists.

### 3. *ModFact*: An Implementation of MDA Standards

Our tool, *ModFact*, can be seen as a user models repositories builder [7]. It takes the definition of user models in input (i.e. the meta-model) and then builds the user models repositories. This generation of repositories is in conformance with the MOF and JMI standards. Thus, these repositories have a standard API defined either in IDL or in Java, which makes them usable, as they provide a user with facilities to create and browse through models. We also generate DTD in accordance to the XMI standard in order to export models to XML.

To achieve these functionalities, *ModFact* is structured into two parts: a **MOF Repository** and the **Engine** (Figure 2). The MOF Repository provides storage and manipulation of MOF meta-models. It is used by the Engine to build the user models repositories. The Engine provides a set of services, namely the generation of IDL interfaces, JMI interfaces and DTD, which are needed for building the user models repositories and for enabling him/her to navigate in the models.

To sum up, *ModFact* is both a user models repository builder and a user models repository.

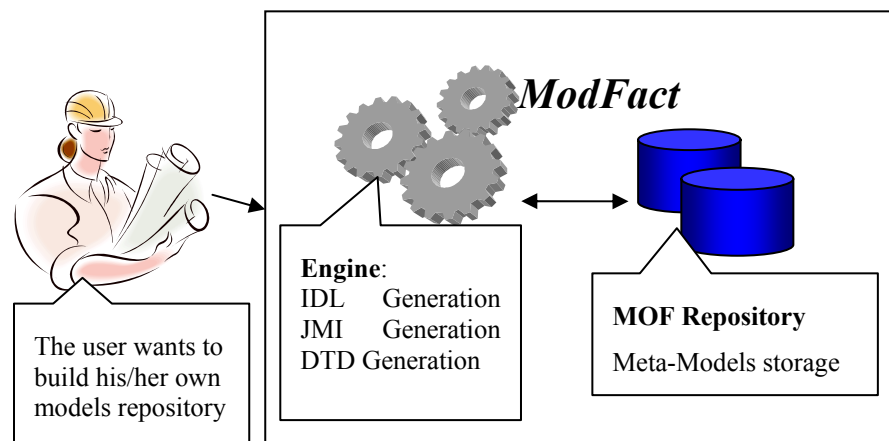


Figure 2: The *ModFact* MOF Repository & Engine

#### 3.1. The Engine

The Engine provides a *ModFact* user with services resulting from the implementation of MDA standards. These services are described hereafter as the “Standards Services”. In addition, it provides some facilities, not defined in the standards, which we identified as required in order to be able to offer a full functionality of models repository construction. We call them the “Value-Added Services”.

##### 3.1.1. The Standards Services

**The IDL Generation Service:** this service enables the user to construct his/her own models repository. It results from the implementation of the part of the MOF standard dedicated to the mapping of meta-models onto IDL interfaces. The *ModFact* module that performs this generation of IDL interfaces inputs packages that present a meta-model, and outputs an IDL module containing the generated IDL interfaces. In our opinion, the easiest way to provide a models repository when having IDL interfaces is to supplement them with their implementation classes in order to get a CORBA repository. In fact, these interfaces together with their implementation classes constitute a user models repository where the elements of a model are CORBA objects that can be handled through this IDL API. Of course the provision of these implementation classes is not supported by the MOF standard as this must be independent from any implementation choice. We will present in the next section the value-added service we developed to generate automatically these classes.

**The JMI Generation Service:** this service is very close from the previous one, but rather than storing models as a set of CORBA objects that a user can handle through IDL interfaces, objects and interfaces are Java ones. The service results from the implementation of the JMI (Java Metadata Interchange) standard. This defines a dedicated mapping of MOF meta-models onto Java interfaces. Thus the *ModFact* module performs the same functionality as the one described above, except that it outputs a Java Jar

containing the generated Java interfaces. As well as the MOF standard, the JMI standard does not support the provision of the corresponding implementation classes. We also have developed a value-added service to generate them automatically.

**The DTD Generation Service:** this service enables the user to write a model in terms of an XML file. To this end, it provides him/her with a DTD that describes the models structure. This DTD is generated from a meta-model provided by the user. This service results from the implementation of the XMI standard, which allows the generation of DTD from any meta-model. The *ModFact* module that performs this generation of DTD inputs packages that present the meta-model, and outputs a file containing a DTD. Thus having such a DTD, the user can write XML files. This means that models are XML files.

### 3.1.2. The Value-Added Services

**The Implementation Classes Generation Service:** As mentioned above, the generated interfaces, either IDL or Java, need to be supplemented with their implementation classes in order to be fully operational. This service provides such an implementation thanks to the automatic generation (in Java) of the implementation classes of the IDL interfaces and of the Java interfaces. To this end, we have defined a set of rules that we called "Templates" that allows the generation of classes from any meta-model. These templates have been very useful for our own development, since we made use of them when generating the implementation classes for the MOF repository (see Section 3.2).

**The MOF API Extension:** Our experience on the use of the MOF API to build a repository, namely the MOF repository, has shown that several useful functionalities required when browsing through models are not directly supported. Let us report some examples:

- This API provides no means to list directly all the `Classes` contained in a `Package`. When needed, the process is to list all the `ModelElements` contained in this `Package` and then to filter them by their types.
- From a `Package`, it is not possible to obtain directly the `Imported Package`.
- This API provides no means to list directly all the `Attributes` of a `Class`. When needed, the process is to list all the `ModelElements` of this `Class` and then to filter them by their types. The same applies for `Operations` and `References of Classes`.
- From a `Class`, it is not possible to list directly some specific `Features` according to a specific criteria (e.g., the list of the `public Features`).
- From an `Association`, it is not possible to obtain directly the `AssociationEnds` that compose it.

Thus we extended the standard API in order to provide enhanced facilities. Here are some examples of operations we implement:

- Packages operations: `classesOfPackage (package) , associationsOfPackage (package) , ...`
- Classes operations: `getAttributes (class) , getAllAttributes (class) , ...`
- Association operations: `associationEnds (association) , ...`
- Packages operations: `ClassesOfPackage (package) , associationsOfPackage (package) , ...`

These facilities are provided in terms of classes named `Helper` we developed (see <http://modfact.lip6.fr> for more details). They enable the user of the MOF repository to handle the repository elements with a high level API.

All these services allowing the user to handle models require as input a meta-model, which these models are compliant with. This meta-model must be provided by the *ModFact* user as an XML file since *ModFact* currently does not include any graphical interface to write or draw meta-models. On the other hand, *ModFact* provides the user with a supplementary service enabling to the export and import meta-models as described hereafter.

**The MOF Meta-models Importation and Exportation Service:** this service enables the exchange of meta-models as XML documents. These must be valid to the standard MOF DTD specified in the XMI

standard. In this way, *ModFact* is open, in the sense that it is capable to import and export meta-models elaborated by other tools, especially CASE tools. However, there exist differences between XMI implementations provided by tools, and this explains why the exchange of MOF meta-models is not so easy. To this end, and in order to facilitate the elaboration of meta-models by using an existing CASE tool, we have developed a module that allows the import of the XMI documents produced by Rational Rose<sup>1</sup> and corresponding to MOF meta-models elaborated with this tool. We chose this CASE tool because it is widely used by designers and it provides a rich graphic interface, which makes easier to draw MOF meta-model as UML diagrams.

### 3.1.3. Summary

To sum-up, through the use of these *ModFact* services, the user creates his/her own models repository as follows:

1. He/she provides the meta-model of his/her models, either as an XML file or through the use of the Import service from the Rose CASE tool.
2. The meta-model is then stored in the MOF Repository of *ModFact*
3. The user makes use of a generation service, namely the IDL, the JMI or the DTD service, in order to get respectively a CORBA repository (i.e., the IDL interfaces with their implementation classes), a Java repository (i.e., a set of Java interfaces with their implementation classes) or an XML repository (i.e., a DTD).

As meta-models need to be stored in order that the *ModFact* engine makes use of them, *ModFact* then includes a MOF repository described hereafter.

## 3.2. The MOF Repository

The MOF repository stores meta-models and permits their manipulation. To create it, we made use of our implementation of the MOF standard, namely the IDL API generation service with the implementation classes generation service. Actually, the mapping defined in the MOF standard and enabling to build an API of models repository from a meta-model can be used as well to build a meta-model repository. In other words, this mapping of a meta-model onto IDL interfaces can also be applied to the MOF model itself due to the fact that the MOF model is auto-descriptive. Thus the algorithm, i.e., the mapping applied to the MOF model outputs a set of IDL interfaces enabling the MOF meta-models manipulation. Here again, supplementing these IDL interfaces with their implementation classes allows building a CORBA repository, but in this case, a MOF meta-models repository. Thus our MOF repository stores meta-models as CORBA objects. Consequently it is a CORBA server.

We already mentioned that the MOF standard does not impose any implementation. Our choice of CORBA has several advantages. The first advantage is simplicity as already mentioned. The MOF API being defined in IDL; the construction of the repository consists in developing the implementation classes. The second advantage is to benefit of the CORBA services. For example, if we want to add a layer of security in the repository, we can use the CORBA security service. Likewise, the persistent service allows obtaining a persistent repository. We currently work on the *ModFact* persistence according to this service.

Finally, let us mention that in a recent release of *ModFact*, we experimented the construction of a MOF repository by using the JMI generation service. More information on this experiment is available on the *ModFact* web site.

### 3.3. On the Use of *ModFact*

*ModFact* is currently available at <http://modfact.lip6.fr>. It is organized as a set of projects, namely the **Repository Project** and the **QVT Project**.

The **Repository Project** deals with the repositories builder. It enables the construction of repositories, either a CORBA repository or a JMI repository. The construction of a CORBA repository can be achieved as described in previous sections, by applying the service of IDL and implementation classes

---

<sup>1</sup> Rational Rose is CASE Tool produced by Rational Rose® software for the modelling and the development of systems using UML.

generation. The resulting repository is a set of CORBA objects. The construction of a JMI repository is quite similar, by applying the service of JMI and implementation classes generation. The resulting repository is a set of Java objects.

In both cases, code is available for downloading. In addition, these services are available online, with no downloading.

The *QVT Project* deals with the models transformation. Although this is out of the scope of this paper, let us just mention that we experimented a simplified version of TRL (Transformation Rule Language), which is a proposal submitted in response to the OMG RFP MOF 2.0 Query/Views/Transformations [8][9]. Thus we provide a service of models transformation, available online.

In the following, we focus our discussion on the analysis we draw from our MOF implementation experiment.

## 4. The MOF Standard Analysis

From our experiment, we learned several lessons regarding the MOF standard, which is a key standard for MDA. Currently, this standard is composed of two parts, each of them addressing a specific issue.

One part is dedicated to the definition of a meta-meta-model, the so-called MOF model, which enables the elaboration of meta-models. This part is very fundamental as having a standardized meta-meta-model is very useful.

The other part is the definition of rules enabling the mapping of meta-models onto IDL interfaces. This aims at building repositories of user models. From a MOF meta-model, a user can obtain the set of corresponding IDL interfaces. These constitute an API through which he/she can handle models in conformance with the meta-models, provided that he/she has developed the implementation classes corresponding to these IDL interfaces.

In our optic, this part of the MOF specification devoted to the definition of a standard API enabling the engineer to create and to browse their models is debatable. We identify two points of discussion. The first one relates to implementation issues and architectural limits we encountered during the development of our tool. The second concerns the soundness of a standardization step for such an API. We develop hereafter these two points.

### 4.1. Pragmatic Issues Related to Implementation

As mentioned in Section 3.1.2, our experiment leads us to analyze that the API defined in the MOF standard is not fully ready for implementation. We identify at least three kinds of gaps, namely the storage choice, architectural issues and the API completeness.

**The Storage Choice:** As any standard, the MOF standard prescribes no choice for implementation. Any storage can be used, such as a file system, a database, or, of course a CORBA server. However, according to this choice, before having a repository, more or less work has to be done. A tool providing a user with the storage of his/her models in a database together with their handling according this standard API seems very complex to develop! We made the choice of CORBA because of simplicity. However, to be able to provide users with CORBA repositories, we had to integrate in our tool the automatic generation of implementation classes.

**Architectural Issues:** the MDA standards devoted to the repositories building, MOF as well as JMI standard, only define how to build the API of users models repository, but they do not provide any recommendation for building such repositories. Although it is not the role of a standard to describe very precise implementation details, however these standards lack prescription as for their implementation. Thus none of them deals with non-functional properties such as persistency, security or performance. In particular in the MOF standard, the API definition in IDL raises a granularity problem. In this approach, each element of a meta-model (e.g., package, class, or instance attribute) is translated into a CORBA object. Depending on the meta-model complexity, the number of objects can be very high. To face this, issues such as performance, persistency and the load increase must be addressed, what the standard does not do. To deal with these non-functional properties, *ModFact* being accessible on the web, we have had to change our code several times and we still improve it.

**The API Completeness:** As mentioned in Section 3.1.2, our experiment led us to extend the standard API as it does not fit in the requirements when browsing through models. Thus it appears that this API is too low level<sup>2</sup> to be really efficient, and needs to be supplemented with operations of a higher level. However it seems to us that this cannot be done in a standardized way. Actually, an API of a meta-model outputs from a meta-model and the IDL generation rules. These must apply to any meta-model. Now providing a high level API that offers methods to browse the whole structure of the meta-model requires knowledge of the meaning of the meta-model elements and of their relations. This means that such a navigation depends on the considered meta-model and cannot be generalized. Thus the standard only deals with a structural view focused on the element being translated. This means that a rule that applies on a meta-class of the meta-model only has the view of the other meta-classes that are directly related to it. If this meta-class is also linked to another meta-class in an indirect way, the rule does not apply, i.e., the corresponding method will not be created in the API.

Analyzing in depth this problem of low level API led us to wonder whether the MOF standard tackles the right level of standardization. We debate this question hereafter.

#### 4.2. On the Soundness of a Standardized Step for an API of Repositories

Among the various MDA standards, the MOF specification is a fundamental one as it deals with the repositories issues through the provision of an API. Now in the MDA approach, the storage of models and thereby the repositories play a central role. Having an API enabling to browse through the models is therefore required. Standardizing such an API appears sensible. It will enable to use the same interface to access models whatever the repositories that store these models. But what is the right level of standardization of such an API? In the MOF standard, the proposed API is low level. It seems to us that this is not the right level, as we illustrated it in the previous section. We base our opinion on a comparison with another area that addresses storage issues too, namely the DataBase Management Systems (DBMS). Let us focus for example on Relational DataBase Management Systems (RDBMS). Many RDBMS exist, each of them having its own solution to store data and to navigate in these data. These are proprietary solutions. What is standardized is the language that enables the user to manipulate data, namely the SQL (Standardized Query Language). It constitutes a high level API that offers facilities in order that the user can easily formulate his/her requests and browse through their data schema. To this end, each RDBMS implements this API and is able to translate these SQL requests into its proprietary low level language in order to perform the queries. Whatever the RDBMS and its internal operation, the user expresses his/her requests in the same way.

In our view, repositories of models should be thought in the same way. The attention must not be placed on the low level API, which depends on the internal mechanisms of the storage means. Such an API cannot be useful to engineers that handle models. Moreover, having standardized repositories is not so useful. Rather, one should stress on the standardization of a high level API, which would provide the engineers with a real service of navigation through the models, whatever the underlying repository.

Let us notice that the MOF 2.0 standards family still includes the definition of rules for the mapping of meta-models onto IDL3.0 interfaces in order to provide an API. However, this does not solve the presented issues, rather the use of IDL3.0 increases complexity in terms of number of generated objects. In fact, the OMG is currently interested in the definition of a query language for models. This is one of the issues of the RFP MOF 2.0 Q/V/T [9]. This is a first step towards the provision of a high level API.

### 5. Conclusion

This article has reported our experiment when implementing some MDA standards and the analysis that we draw from this experiment. We make a specific focus on the MOF standard since it defines an API in IDL to handle models. We describe the tool we developed, named *ModFact*. This implements the MOF, XMI and JMI standards. In this way, it enables a user to build his/her models repository as a set of CORBA objects, or as a set of Java objects or as XML files. Moreover, the use of the XML format enables the exchange of models. During our development, we faced some issues not considered in these standards. Thus we had to provide value-added services in order to reach this goal of providing storage and exchange functionalities.

---

<sup>2</sup> “Low level” means that some API operations are very elementary and the user must add some processing in order to get an efficient navigation. See examples mentioned in Section 3.1.2 that illustrate this point.

This experiment enables us to conclude that the current MDA standards completeness is debatable. Especially, the MOF specification does not tackle the right level of standardization with respect to the provision of an API for the models handling. Software engineers, who apply an MDA approach when developing software, need a higher level API. The new RFP MOF 2.0 Q/V/T issued by OMG aims at fulfilling this gap.

Beyond this storage functionality and the provision of an API, which are fundamental for the MDA, other issues are still open. Producing software according to an MDA approach requires a set of various tools, each of them performing its task based on models handling. For example, a CASE tool is required to create and edit models; a translation engine is required for models transformation; a repository for models storage; and so on. In order to constitute a software production chain, all these tools will have to interoperate. We identify that this requires a well-defined architecture of a platform supporting this integration. We are currently working at the definition of such a platform. Our approach is service-based, that is the identification of an interface for each tool describing the service provided by this tool (see [11] for more information).

## References

- [1] OMG, “Model Driven Architecture (MDA)”, OMG ormsc/2001-07-01, July 09-2001.
- [2] D.S. Frankel, “Model Driven Architecture – Applying MDA to Enterprise Computing”, OMG Press, Wiley Publishing, 2003
- [3] OMG, “Unified Modeling Language: Superstructure, version 2 alpha R3 (Draft)”, OMG, July 17 2002.
- [4] OMG, “Meta Object Facility version 1.3”, [www.omg.org](http://www.omg.org), november 2001
- [5] OMG, “Common Warehouse Metamodel (CWM)”, OMG ad/2001-02-01, January 2001.
- [6] OMG, “XML Metadata Interchange (XMI) Specification v1.2”, January 2002
- [7] Sun, Java Community Process, “Java™ Metadata Interface (JMI) Specification”, <http://java.sun.com/products/jmi>, june 2002
- [8] The *ModFact* Project, <http://modfact.lip6.fr>
- [9] Alcatel, Softeam, Thales, TNI-Valiosys, « Response to the MOF 2.0 Query /Views /Transformations Initial Submission », [www.omg.org](http://www.omg.org), march 2003
- [10] OMG, Request for Proposal MOF2.0 Query /Views /Transformations, ad/2002-04-10, [www.omg.org](http://www.omg.org), April 2002
- [11] <http://www-src.lip6.fr/projets/racine/odac-eng.html>