João Paulo Andrade Almeida
Luís Ferreira Pires
Marten van Sinderen (Eds.)

Milestones, Models and Mappings for Model-Driven Architecture

European Workshop on Milestones, Models and Mappings
for Model-Driven Architecture (3M4MDA), Bilbao, Spain,
July 11, 2006
Proceedings

Preface

This volume contains the proceedings of the European Workshop on Milestones, Models and Mappings for Model-Driven Architecture (3M4MDA) held on 11 July 2006 in Bilbao, Spain, in conjunction with the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006).

The 3M4MDA 2006 workshop aims at helping the convergence of the research and practical application, by focusing on the milestones in the model-driven development process, the models to be used, and mappings to be established between these models. Many issues have risen from the practical application of the model-driven approach, most notably those that relate to the purpose and the characteristics of the system under design and that translate to requirements on milestones, models and mappings to be applied in the model-driven development process. These issues should be taken into account in the next generation of model-driven methods and modelling techniques.

There have been 14 paper submissions to this workshop, and 8 papers have been accepted for publication and oral presentations. All selected papers are of high quality, thanks to the professionalism of the authors, reviewers and program committee members.

The papers in this volume are representative for the current research activities on the topics indicated by the title of the workshop, namely *milestones*, *models* and *mappings*. These papers have been grouped around these topics in the three sessions of the workshop.

We would like to take this opportunity to thank the people who have contributed to the 3M4MDA 2006 workshop. We wish to thank all authors and reviewers for their valuable contributions, and we wish them a successful continuation of their work in this area. Finally, we thank the organization of the ECMDA-FA 2006 conference in which this workshop has been embedded.

June 2006

João Paulo Andrade Almeida
Luís Ferreira Pires
Marten van Sinderen

Organisation

Workshop Chairs

João Paulo Andrade Almeida	Telematica Instituut, the Netherlands
Luís Ferreira Pires	University of Twente, the Netherlands
Marten van Sinderen	University of Twente, the Netherlands

Programme Committee

Jan Aagedal	SINTEF, Norway
Dave Akehurst	University of Kent, UK
Colin Atkinson	University of Mannheim, Germany
Jan Bank	Compuware, the Netherlands
Mariano Belaunde	France Telecom R&D, France
Rolv Bræk	NTNU, Norway
Remco Dijkman	University of Twente, the Netherlands
Keith Duddy	SoftMetaWare, Australia
Slimane Hammoudi	ESEO, France
Maria-Eugenia Iacob	Telematica Instituut, the Netherlands
Olaf Kath	IKV++, Germany
Anneke Kleppe	University of Twente, the Netherlands
Peter Linington	University of Kent, UK
Dick Quartel	University of Twente, the Netherlands
Richard Soley	Object Management Group, USA
Maarten Steen	Telematica Instituut, the Netherlands
Antonio Vallecillo	University of Málaga, Spain

Supporting Organisations

Centre of Telematics and Information Technology, University of Twente
Telematica Instituut
Freeband A-MUSE project
Object Management Group

Table of Contents

Towards a new software development process for MDA

Abdessamad Belangour[1], Jean Bézivin[2], Mounia Fredj[1]

[1] Laboratoire de Génie Informatique, ENSIAS, University Mohammed V, B.P. 713, Souissi
Rabat, Morocco.
belangour@yahoo.fr
fredj@ensias.ma
[2] Atlas Group, INRIA and LINA, University of Nantes, 2, rue de la Houssinière –
BP 92208, 44322, Nantes Cedex 3, France.
Jean.Bezivin@univ-nantes.fr

Abstract. In this paper we propose a new software development process which is called M2T (MDA[TM] 2 Tracks). The M2T process provides an implementation of the MDA[TM] approach while relying on a Y shaped development cycle. The left branch of the Y cycle corresponds to Platform Independent Model (PIM) while the right one corresponds to what we call an explicit Platform Description Model (PDM) representing the targeted platform. The evolution of the two branches (i.e. PIM and PDM) corresponds to a series of transformations. The combination of the PIM and the PDM is then driven by a fixed Design Decision Metamodel (DDM) that captures the mapping features between both upper branches of the "Y" to produce a the Platform Specific Model (PSM). The adoption of the MDA[TM] as a guideline for the M2T process made its approach different from known methods but closer at the same time.

1 Introduction

During its brief history, information technology has produced a huge amount of notations and methodologies that aim to handle information. Each wave of notations arises, most of the time, after a major software crisis that shakes the software development community. Shortly after, it proves its inability to overcome the increasing complexity of information systems. One of the major sources of problems is that these notations and methodologies were used, at best, as a visual guide that is rarely involved industrially in the development process. Until the 70's for example, software developers viewed software development as an artistic venture that failed, using individual creative operations, to conquer the system's acceleration of complexity [1]. Thus, the software community, represented by the OMG, undertook a series of steps to overcome this crisis. The latest step is the MDA[TM] (Model Driven Architecture) initiative launched in 2000 [2]. The base principle of the MDA[TM] is the development of platform independent models (PIM) capitalizing the business functionalities of a system and their transformation to platform specific models (PSM) for a given techno-

logical platform (Java/EJB, C #/.Net, etc.). The MDA^{TM} relies on a solid base of standards like UML [3] [4] [5], MOF [6] and XMI [7]. However, to be widely accepted, it needs to fix the following issues:

- Absence of a clear transformation approach that leads to PSMs starting from a given PIM.
- Absence of a model representative of the targeted platform in the approach.

In a previous paper [8] we proposed an approach that fixes these issues. In this paper our goal is to take this approach further by transforming it to a software development process. Thus, we propose a new software development method (M2T method) which is based on the Y shaped development cycle similarly to the 2TUP method of P.Roques [9] which is an implementation of the UP (Unified Process) process [10]. The M2T borrows the 2TUP's phase of analysis but it is completely different from it. To demonstrate our approach, we rely on a small illustrative use case.

This paper is organized as follows. In section 2 we introduce the M2T process with its different phases. In section 3 we apply it to an illustrative case study. Finally, in section 4, we conclude with remarks and future perspectives.

2 Proposition of the M2T process

The M2T (short for MDA^{TM} 2 Tracks) process is a software development process which is partially based on the 2TUP process as it shares with it its reliance on the Y shaped cycle. Besides, M2T is totally different from the 2TUP process as it has integrated the MDA^{TM} process as its backbone while the 2TUP still mixes business and platform features. Thus, the M2T process shares the analysis phase with the 2TUP as a starting point but follows a totally different approach as we show in Fig. 1.
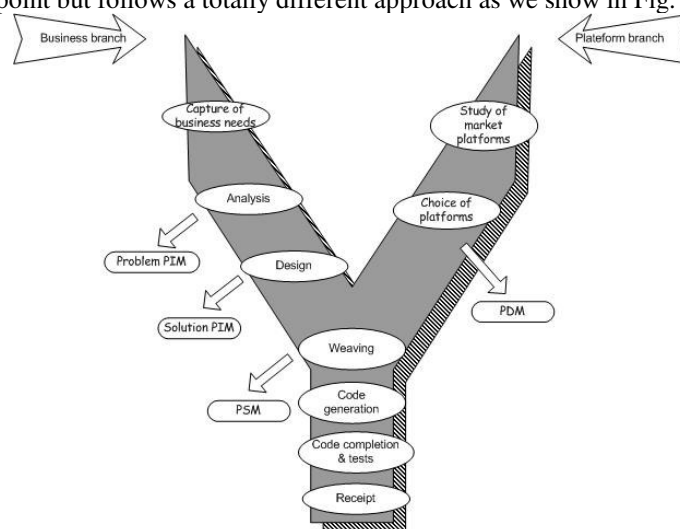

Fig. 1. The Y development process of the M2T method

The approach adopted by the M2T combines the classical software development phases with the MDA^{TM} PIM-to-PSM transition. Since the earlier stages of the 2TUP and the M2T processes are the same, we will go directly to the design phase. Note that the only difference on this level is that the analysis phase ends with a PIM that describes the problem and that we called problem PIM.

2.1 Proposition of the solution : the design

The goal of this phase is to propose a solution to the problem described by the problem PIM while staying independent from the platform. In order to build this solution, we propose to proceed according to the following stages. We recommend starting with the package on which all the other packages rely. The first stage consists in gathering all the dynamic diagrams (sequence diagrams) developed in the analysis phase that are related to the package in process. Then, each of these diagrams must be completed by detailing the different scenarios it represents. The completion of these scenarios consists of adding details of realization but always in independent platform context. This time, these scenarios must contain the totality of the messages exchanged between the various objects from the collection of data until the return of the results. The detail of these scenarios requires certain operations with a non-business nature which leads to additional solution features (classes, interfaces, etc..) that accompany the business features to provide the whole expected functionality of the system. Consequently, the solution arises as a global package that is essentially defined around a business subpackage together with a set of additional subpackages that operates on it to deliver the required functionality. Therefore, the solution package is composed of the following subpackages (Fig. 2):

- A Business subpackage containing the problem PIM resulting from the analysis phase. Note, it can possibly undergo changes within this stage.
- A Presentation subpackage containing the GUI classes that will be used to communicate with the user. They should not be too detailed by the designer as they are going to be replaced by those of the platform.
- A Utility subpackage containing the utility classes for recurrent processing. They should not be too detailed here either ; same for the presentation classes, and for the same reasons.
- A Persistence subpackage for reading and storing data from/to a data source such as data bases.
- An Exception subpackage for exceptions processing that throws the possible errors related to the handled classes.
- A Boundary subpackage for boundary classes [11] [12] that form a mediator or an intermediary for data exchange between the presentation subpackage and the core package.
- Finally, a Core subpackage which is the core of the solution relying on the other subpackages to propose a global solution to the exposed problem.
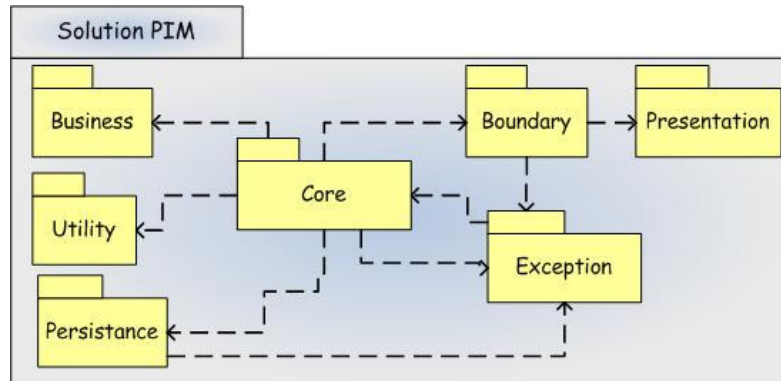
Fig. 2. Solution composition

Consequently, the logical steps or stages of the design phase are therefore as follows:

1. For each scenario resulting from the analysis.
    a. Determine data to get from the user or to display to him.
    b. Add corresponding presentation classes.
    c. Add boundary classes that communicate with presentation classes to get data from user or to display results.
    d. Detail the solution from the retrieval of data from boundary classes until the completion of the functionality expected by the user.
    e. When a required operation can be described as common, create a utility class to contain it and place it in the package of utility classes.
    f. Each access to a data from a data source must be delegated to the persistence subpackage classes.
2. Detail the exception scenarios the same way, to deduct the classes of exceptions that form the exceptions subpackage.

It is important to underline that the presentation, utility, persistence and exception subpackages should not be too detailed because this is not the goal of the system under development and because they are well developed at the platform level. It can, therefore, form a part of a reusable PIM library. On the other hand, the business solution must be very detailed owing to the fact that it is a proprietary solution aiming at achieving the goal of the system under development and provide the expected functionality.

At the end of this phase, we end with a solution PIM model which contains all the "ingredients" of the solution while being independent of any platform. This PIM is then ready to be integrated into a specific platform to become a PSM for the selected platform. This will be explained and shown in the next section.

2.2   The weaving with the platform

The analysis and the design in our method are activities of the left branch of the Y development cycle we follow. The activity of the right branch of the Y cycle in our method M2T is limited to the choice of the applicative platform that the solution PIM will be bound into after a study of available platforms in the market. If the solution PIM is composed of several components, each component may be implemented on a different platform. The choice of a given applicative platform consists of having models that represent them in compliance with their respective metamodels. The CORBA platform and the Java /EJB platform, for example, have standard metamodels. In this case, the effort must be concentrated on the search (or even development) for tools that offer capabilities for extracting models from these platforms. That is why we have proposed a metamodel and a tool to extract models for the .Net platform [8]. The activity of the right branch of the cycle in Y is consequently reduced to the availability of models of the chosen platforms. We called such models PDM (Platform Description Model) and we made their existence a pre-condition to the transformation of the PIM into PSM in our implementation approach of the MDA^{TM}. We also made it a basic stage in our M2T software development process.

The weaving phase of the PIM model with a PDM representing a given platform, or more exactly a view on a given platform, is articulated around a model which captures the design decisions and which we called DDM (Design Decision Model). This model which is described by a DDM metamodel defines these design decisions as a set of mappings that connects entities belonging to the PIM on the one hand, with entities belonging to the PDM on the other hand.

The weaving phase occurs in two stages:

1. Definition of the design decision model (DDM): It consists of building a model of decisions of design which conforms to the metamodel of DDM. The DDM model allows linking the PIM constituent with the PDM features. These links can be, according to DDM metamodel [8], replacement, use or extension links, etc.
2. Execution of the transformation: this stage aims to generate the PSM of the application, by carrying out the automatic weaving of the PIM and the PDM. This weaving is done while being based on the DDM defined in the preceding stage.

DDA (Design Decision Assistant) [8] is a tool that we developed for this reason. It allows the definition of the DDM and the generation of the PSM by executing the defined transformation which can also be carried out using the recent languages of transformation like ATL [13] or YATL [14] or even the traditional language of transformation XSLT [15].

The code generation can be realized by loading the PSM in a CASE tool that supports the targeted platform while the remaining phases (code completion & tests and receipt) are classical.

3 A case study

In this section we are going to illustrate how our M2T process works through a simple illustrative case study that will be implemented in the C#/.Net platform. For simplification we will revisit a previous case study [8] that we have modified.

3.1 The problem

We aim at elaborating a small query engine that performs advanced searches for document titles (books, papers, journals, etc.) from a library database. This query engine will be developed for C#/.Net. We then have the choice to make it as ASP.Net application, a console application or a Windows application since all those types of applications are supported by this platform. The user queries rely on logical operators for a better description of the search. The syntax for queries (expressed in BNF) is the following one:

```
– Query = Expression "?"
– Expression = Term {"OR" Term}
– Term = Factor {"AND" Factor}
– Factor = Word | "(" Expression ")" | "NOT" Factor
– Word = Letter {(Letter |Digit)}
– Letter = UpperCase | LowerCase | Underscore
– Uppercase = "A".."Z"
– Lowercase = "a".."z"
– Underscore = "_"
– Digit = "0".."9"
```

3.2 Capture of business needs

In this phase the system is considered as a black-box to focus on its expected functionalities. We should build the dynamic context diagram and thus we need to determine the actors of the system and the messages they exchange with it. The user querying the system is an obvious client of the system. We could also talk about the librarian that feeds the database with entries or updates it but this aspect is outside the scope of the limited engine we took as an example. Consequently the system has an only one actor.
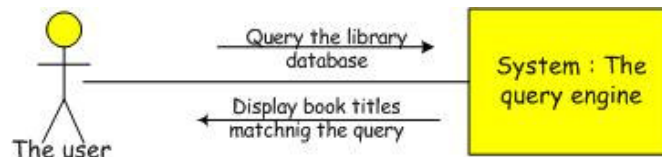

Fig. 3. Query engine dynamic context diagram

The next logical step is listing all the use cases and their description. The illustrative case study we have for example is fairly simple; we have only one use case with one key actor and no secondary actors.

Use case	Key actor, secondary actors	Sent/received messages by the actor
Query the library database	The user	Emit : search expression Receive : list of matching titles

The description of the use case above looks like:

Query the library database (the user): o   **Intention**: perform an advanced search o   **Actions**: type a query expression and validate it.

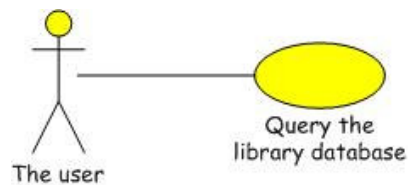The graphical representation of this use case is as follows:

Fig. 4. Query engine use case

As this case study for demonstration purposes is relatively simple we do not need to add an activity diagram or to clarify a particular complex scenario by a sequence diagram. Moreover, there is no structuring phase as we own just one use case. The complexity of this program comes essentially from the structure of the query. By interpreting the syntax of the query we may add roughly some additional business classes and associations that link them together. Thus the diagram of participating classes becomes as in Fig. 5:
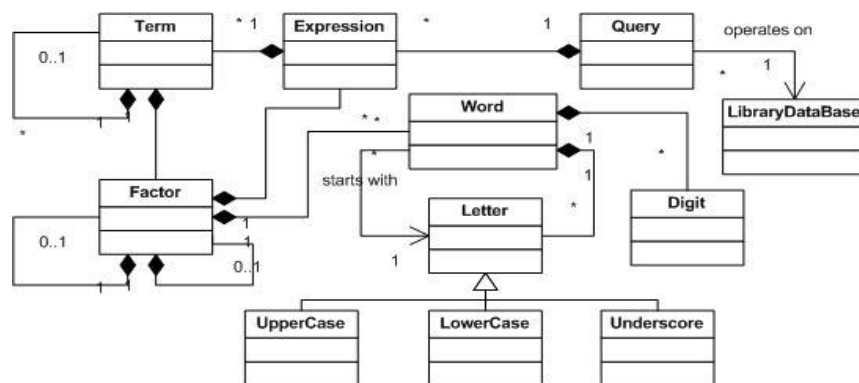

Fig. 5. Diagram of participating classes

The above diagram of participating classes identifies most of the business classes and is going to be refined in the analysis section.

3.3 The analysis phase: towards the problem PIM

In this phase we build an analysis model that corresponds to the problem PIM. We start building the static structure diagram by refining and enhancing the participating classes' diagram. The refinement is achieved by adding missing classes, generalizations, attributes and operations through a first series of refinements. The development of the dynamic diagrams allows us to perform a second series of refinements.

In our case study we have three scenarios. The first scenario is that the user questions the engine which succeeds in finding document titles that match the query within database and displays them. The second scenario is there's no match for the query and a message of apologize is displayed. Finally, the last scenario is that the query is inaccurate which means that its syntax was not respected and that it should be retyped correctly. We will show the first scenario illustrated with the following sequence diagram (see Fig. 6). In this diagram, after the user types the query, the Query processQuery() method is called up. To perform matches, the Query needs to have titles of the documents which are created by the LibraryDatabase class. After examining all the titles, the Query displays the results to the user.
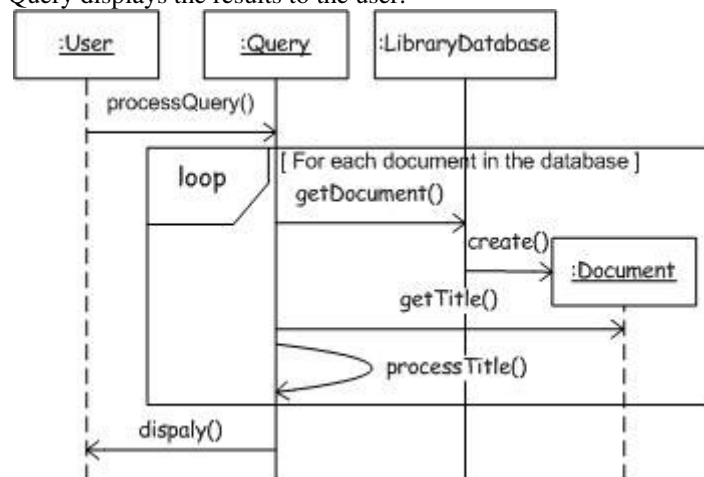

Fig. 6. Sequence diagram of the first scenario

Each class with a rich behavior, like the Query class, for example, needs a state-transition diagram that will help to understand its dynamic as shown in the following figure (see Fig. 7) :
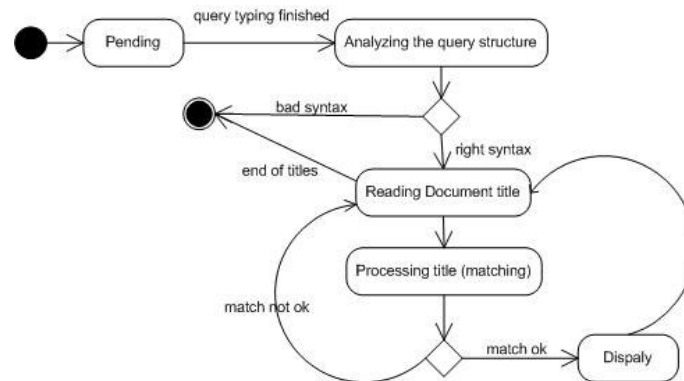
Fig. 7. State diagram for the Query class

One of the great advantages of state diagrams is that it helps us understand the system's dynamic management rules.

After the elaboration of the different dynamic diagrams, comes the stage of their confrontation with the static structure diagram. The availability of the dynamic view together with the static view allows filling any missed gap in the analysis phase. Consequently we perform a new series of refinements that leads to the problem PIM of Fig. 8.

Fig. 8. Static structure diagram of the query engine

The static structure diagram above is the analysis model which corresponds to a problem PIM for the interrogation engine case study. In the next phase, additional features (classes, interfaces, etc.) will be added to complete the final solution that we call solution PIM and that corresponds simply to a PIM in the OMG official terminology.
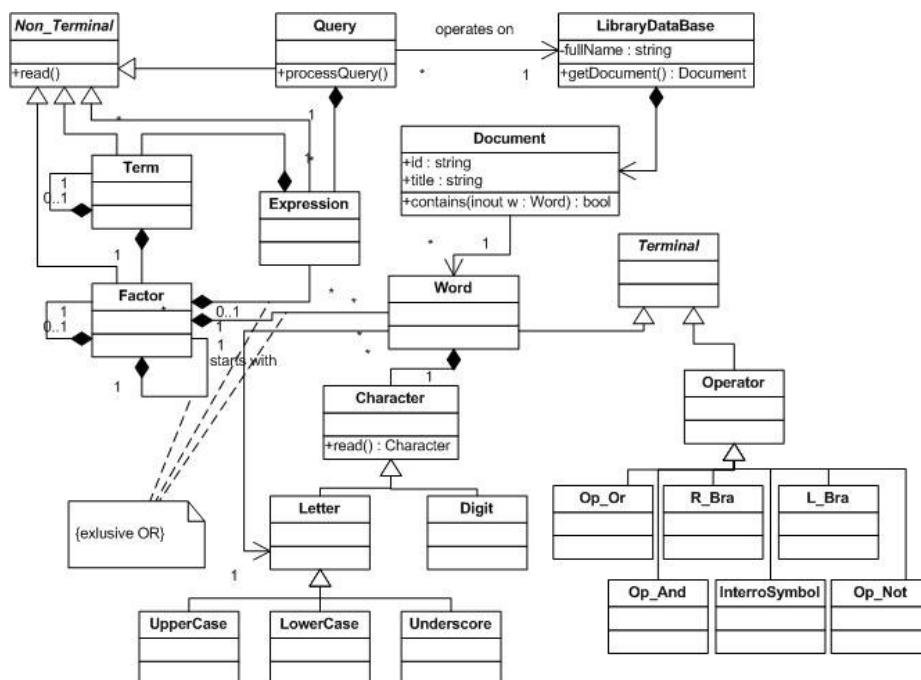
3.4 The design phase: towards the solution PIM

After the formal description of the problem which leads to the Problem PIM we tackle the building of the solution part of the PIM. According to our terminology this part is called solution PIM. The main characteristic of this phase is that it is also independent from the platform.

Step 1: The first step towards the building of the solution, according to our M2T process, is to determine the input and output data that will consequently determine most important features if the GUI. These are as follows:

- Input: query text
- Output: results

After that, we have to determine which GUI classes we are going to use. These classes must encompass the correspondent classes for the inputs and the outputs of the application. A sketch of the GUI gives an idea of those classes, as we can see in the next figure (Fig. 9):
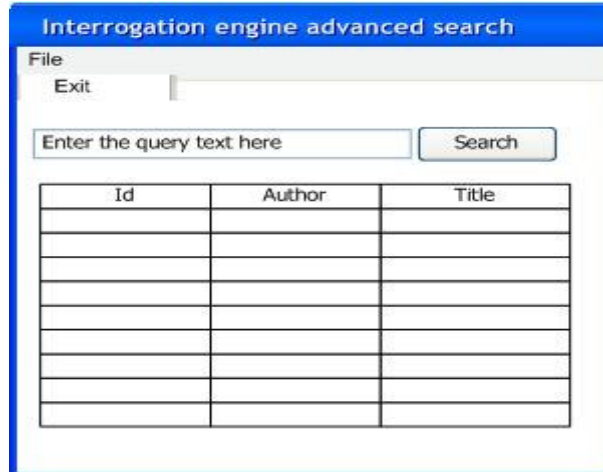

Fig. 9. A sketch of the interrogation engine GUI

We then build the first package that contains these GUI classes with the appropriate operations for each one of them (see Fig. 10).
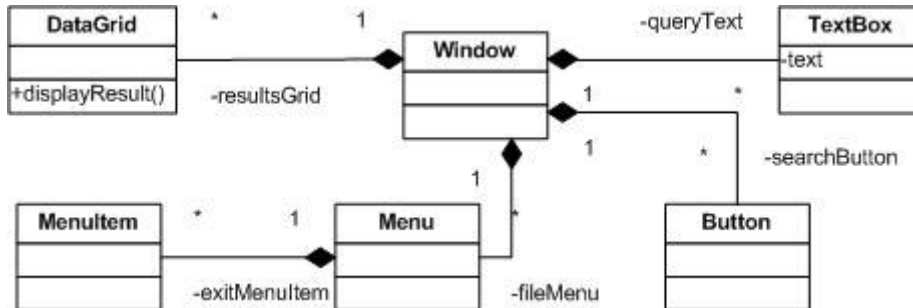
Fig. 10. Presentation package classes

Step 2: After that, we cross to the construction of the boundary classes. Theses classes communicate with the input/output and are as follows:

- SearchTrigger class: the button class will take the string representing the query and transmit it to this boundary class which will pass it to the Query class for process. Notice the addition of the association between the Button and the TextBox classes.
- Result class: this class gets the results of the search and communicates it to the DataGrid class to display it.

Therefore the boundary package contains only these two classes. The presentation classes relate to these classes as in Fig. 11.
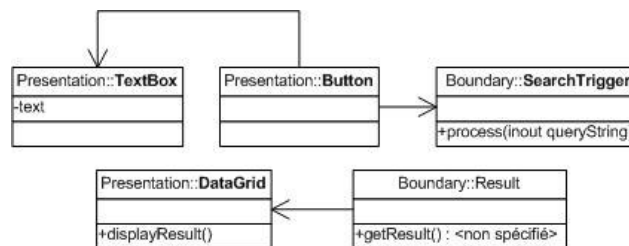
Fig. 11. Relation Between presentation and boundary classes

Step 3: In this step the designer should think how he is going to solve the problem by relying on the behavioral diagrams from the analysis. In his way of doing this, every time he expresses the need for a given class, he must determine its nature and ultimately add it to the adequate package. For our case study, the basic idea behind the solution is the remark that a query is a logical proposition that needs to be interpreted as true when it matches a title of a document or false if not. This causes the adding of a set of classes whose contents are detailed relying on sequence diagrams that space limitations prevent, unfortunately, to detail. Thus, we come to the next solution diagram of the solution PIM in Fig. 12 (without the boundary and the presentation classes).

Fig. 12. Solution PIM

In this solution PIM, we gathered solution classes from different packages to show how they relate to each other but they are actually separated in their respective packages.

3.5 The weaving phase

After the elaboration of the solution PIM which is called simply PIM in the official OMG terminology we move to its binding within a given platform. We have chosen the C#/.Net platform for which we have models, thanks to the PDMgenerator tool [8]. The binding should be carried out in accordance with the DDM metamodel.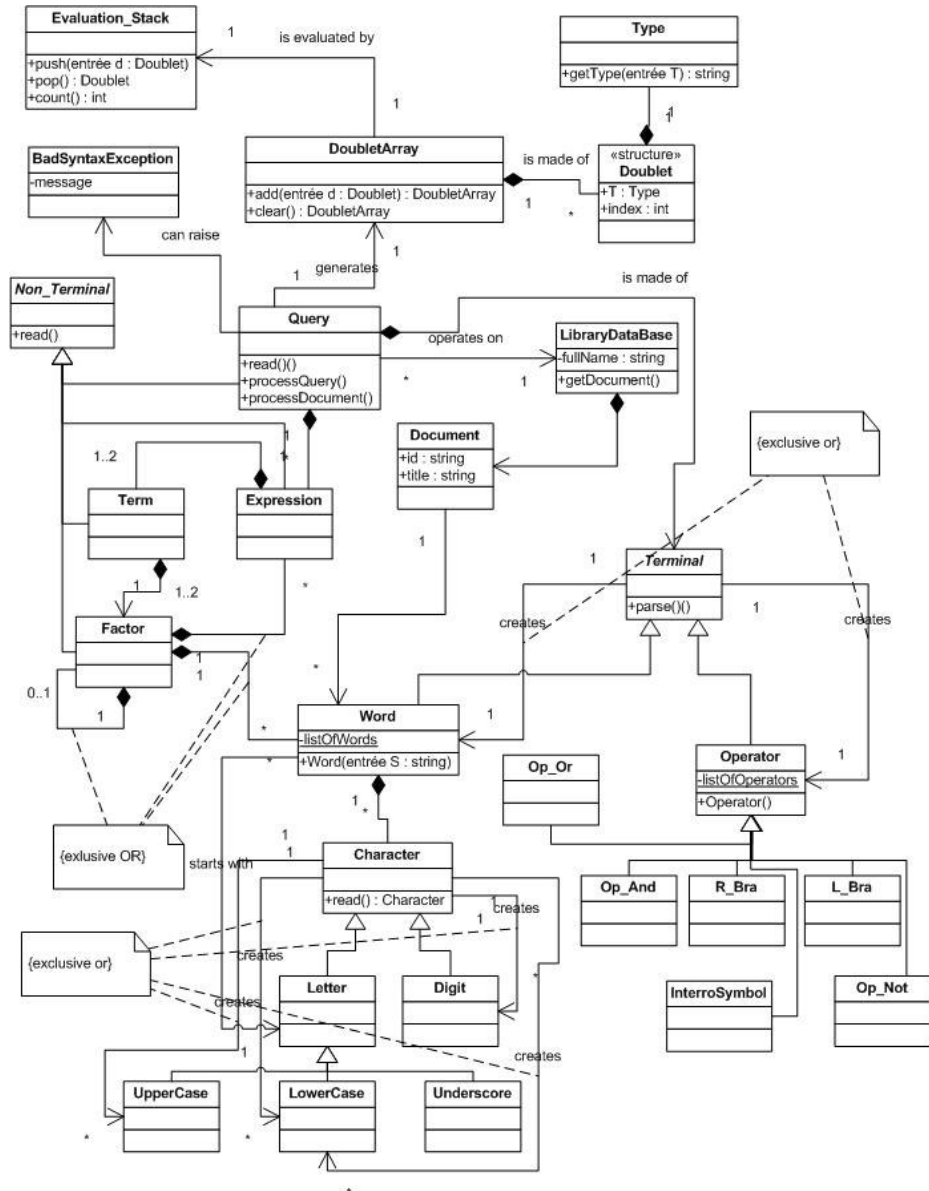 We can again rely on the DDA tool for this purpose. The DDA allows the loading of the PIM and the PDM and the generation of the PSM. The weaving of the PIM with the PDM representing a given platform supposes the user has a good knowledge of the targeted platform to be able to make the good weaving decisions.

For our case study we proceed as follows: we load the PIM within the DDA tool. Then we load the PDM. After that, we start by defining a replacement (*Is_Replaced_By*) mapping for the datatypes as in table 1.

Table 1 - *Data types binding table*

Datatype	Mapping Decision	PDM entity
string	*Is_Replaced_By*	String (class)
integer	*Is_Replaced_By*	Int32 (structure)
float	*Is_Replaced_By*	Single (structure)
bool	*Is_Replaced_By*	Boolean (structure)
char	*Is_Replaced_By*	Char (structure)
short	*Is_Replaced_By*	Short (structure)

Afterwards we tackle the linking of the other PIM elements with the C#/.Net platform features relying on our knowledge of this platform. The database in our case study is an Oracle database for which the .Net class library owns a dedicated namespace: System.Data.OracleClient namespace. Explaining the motivation behind each decision is not the focus of this paper. Consequently, we summarize these decisions in Table 2. Notice that each of the made decisions must carry the name of the decision maker, a date and a motivation according to the DDM metamodel.

Table 2 - *Classifiers binding table*

PIM entity	Mapping Decision	PDM entity
BadSyntaxException	*InheritsFrom*	Exception
Button	*IsReplacedBy*	Button
DataGrid	*IsReplacedBy*	DataGrid
Document	*uses*	String
Doublet_array	*IsReplacedBy*	ArrayList
Evaluation_Stack	*IsReplacedBy*	Stack
LibraryDatabase	*uses*	OracleConnection
LibraryDatabase	*uses*	OracleCommand
LibraryDatabase	*uses*	OracleDataReader
Menu	*IsReplacedBy*	Menu
MenuItem	*IsReplacedBy*	MenuItem

Query	*Uses*	StreamReader
Terminal	*Uses*	StringBuilder
TextBox	*IsReplacedBy*	TextBox
Type	*IsReplacedBy*	Type
Window	*IsReplacedBy*	Form

The next step after defining these mappings, using the DDA tool for example, is the generation of the PSM which is a .Net PSM. This PSM will be accompanied with a DDM file which will keep track of all the decisions made. To generate the code, the PSM can be loaded in any CASE tool that supports the chosen platform. Then it can be completed, tested and deployed as for any classically developed application.

4 Conclusion

In this paper we presented a new software development process: The M2T process. The proposal for this method comes is aimed at filling a vacuum in the software development processes which are in conformity with OMG MDA[TM] initiative. Even though many approaches tried to address this issue they are much more complicated and do not offer a simple alternative [16][17][18]. Ours proposes an approach which shows how to exploit the MDA[TM] which is presently handicapped by the lack of such an approach for its exploitation. Our method shares the analysis phase and the Y development cycle with the 2TUP process but is completely different for the rest of the phases. In our M2T process, models and transformation of models occupy a central place. Each stage is balanced by one or more models which are PIM models in analysis and design phases and PSM models after the weaving with the PDM of the selected platform. As prospects we will develop a plug-in that we may integrate with one of the commercial CASE tools in order to support our method. This will make the task of the project managers easier and will let them concentrate on the projects they are developing from one hand and let them benefit from all the quoted advantages of the MDA[TM] from another hand.

References

1. Grässle P., Baumann H., Baumann P.: UML 2.0 in Action: A project-based tutorial. Packt publishing (2005).

2. OMG : MDA Guide V1.0.1. OMG Document: omg/03-06-01 (2003).

3. OMG : Unified Modeling Language (UML) Specification: Infrastructure version 2.0. OMG Document: ptc/04-10-14 (2004).

4. OMG : Unified Modeling Language: Diagram Interchange version 2.0. OMG Document: ptc/05-06-04 (2005).

5. OMG: Unified Modeling Language: Superstructure version 2.0. OMG Document: formal/05-07-04 (2005).

6. OMG: Meta Object Facility 2.0 Core Specification. OMG Document: ptc/04-10-15 (2004).

7. OMG : XML Metadata Interchange Specification 1.2. OMG Document: formal/02- 01-01 (2002).

8. Belangour, A., Bézivin, J., Fredj, M.: The Design Decision Assistant tool: a contribution to the MDA approach. 1ères journées sur l'ingénierie dirigée par les modèles IDM'05, Paris (2005). Available at http://idm.imag.fr/idm05/documents/16/16.pdf.

9. Roques, P., Vallée, F. : UML 2 en action : De l'analyse des besoins à la conception J2EE. Eyrolles, 3rd edition (2004).

10. Jacobson, I., Booch, G., Rumbaugh, J. :The Unified Software Development Process, Addison-Wesley, 1st edition (1999).

11. Conallen, J. : Concevoir des applications Web avec UML. Eyrolles, 1st edition (2000).

12. Rosenberg, D., Scott, K.: Applying use case driven object modeling with UML – an annotated e-commerce example. Addison-Wesley, 1st edition (2001).

13. Bézivin, J., Dupé, G., Jouault, F., Rougui, J. E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. OOPSLA'03 Workshop on Generative Techniques in the Context of the MDA, California, USA (2003).

14. Patrascoiu, O.: Mapping EDOC to Web Services using YATL. EDOC 2004: 286-297(2004).

15. W3C: XSL Transformations (XSLT) Version 1.0. (1999), http://www.w3.org/TR/xslt.

16. Domain-Specific Languages - An Overview. Available on http://compose.labri.fr/documentation/dsl/dsl_overview.php3.

17. Greenfield, J., Short, K., Cook S., Kent S., Crupi J.:Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools (Paperback). Wiley, 1st edition (2004).

18. Ordstrom, S.: Experiences in Developing Model-Integrated Tools and Technologies for Large-Scale Fault Tolerant Real-Time Embedded Systems. OMG First Annual Model-Integrated Computing (MIC) Workshop, Arlington, VA, USA (2004).

A MDA-based Development Process for Collaborative Business Processes

Pablo David Villarreal[1], Enrique Salomone[1,2], Omar Chiotti[1,2]

[1]CIDISI, Universidad Tecnológica Nacional - Facultad Regional Santa Fe, Lavaisse 610, 3000, Santa Fe, Argentina
(pvillarr@frsf.utn.edu.ar)
[2]INGAR-CONICET, Avellaneda 3657, 3000, Santa Fe, Argentina
({salomone,chiotti}@ceride.gov.ar)

Abstract. The modeling and specification of collaborative business processes is a key issue in order to enterprises can implement Business-to-Business collaborations with their partners. These processes have to be defined first in a business level using a technology-independent language and then in a technological level using a Business-to-Business standard. Both definitions must have a mutual correspondence. To support the above issues, the Model-Driven Architecture (MDA) Initiative has been identified as a key enabler to build a model-driven development method. In this paper, we describe a MDA approach for collaborative business processes. We present the phases, artifacts (models and code), languages, mappings and transformation techniques that make up the development process of this MDA approach. We identify the issues to be taken into account in a MDA approach for collaborative processes, in order to guarantee that the generated B2B specifications fulfill the collaborative processes agreed between the partners in a business level.

1 Introduction

To compete in the current global markets, enterprises are focusing on the setting up of Business-to-Business (B2B) collaborations with their partners. A B2B collaboration implies the integration of enterprises in two levels: a business level and a technological level [22, 23]. In order to accomplish inter-enterprise integration at both levels, one of the main challenges is the modeling and specification of *collaborative business processes*. Through these processes, partners undertake to jointly carry out decisions to achieve common goals, coordinate their actions and exchange information from a common global point of view.

In the business level, partners have to focus on the definition of *collaborative business processes*, which describe the explicit behavior of the collaboration along with the partners' responsibilities. In this level, collaborative processes have to be defined in an independent way of the implementation technology, because business analysts and system designers do not want to deal with the technical details.

In the technological level, enterprises have to focus on the implementation of the B2B information system that supports the collaborative processes. This system is

made up of autonomous, heterogeneous and distributed components owned by enterprises to support the joint execution of collaborative processes. Interoperability between these components can be achieved by using B2B protocols [3]. Two important interoperability aspects are supported by these protocols: the specification of the collaborative processes and the specification of the partners' interfaces that compose the B2B information system, both based on a technological-specific standard language. Currently, there are two main technologies proposed to specify B2B protocols [3]: standards based on Web Services Composition, such as BPEL (Business Process Execution Language) [4] and WS-CDL (Web Services Choreography Description Language) [14]; and standards based on Business Transactions, such as ebXML [20].

Thus, collaborative processes must be defined first in a business level using a technology-independent language and then in a technological level using a B2B standard. However, the development of these B2B solutions is complex and costly [1]. In addition, both definitions must have a mutual correspondence [1, 3]. Another complexity is B2B standards provide or are based on different languages to specify collaborative processes and partners' interfaces, but these standards do not provide techniques to support the transformation or verification of correctness between the collaborative process specifications and their corresponding partners' interfaces. Hence, it is necessary a mechanism that also ensures, in the technological level, consistence between the collaborative process specifications and the partners' interfaces specifications.

To accomplish the above issues, the Model-Driven Architecture (MDA) Initiative [17] has been identified as a key enabler to build a model-driven development method for collaborative processes [21, 23]. In this way, we propose a MDA approach for collaborative processes, which supports: the design of collaborative processes independent of the idiosyncrasies of particular B2B standards; and the automatic generation of specifications based on a B2B standard from conceptual collaborative process models. The main benefits of this approach are:

- Increase of the abstraction level. The main artifacts of the development are the technology-independent collaborative process models.
- Reduction of development time and costs because solutions are generated automatically from conceptual models of collaborative processes.
- Guarantee of consistency in the generated technological solutions because they are generated automatically from a well-defined transformation procedure. On one hand, they are consistent with the processes defined in the business level. On the other hand, the specifications of processes and their corresponding partners' interfaces specifications are also consistent.
- Independence of the collaborative process models from the B2B standards, which increases the reuse of these models.

To support the design of collaborative processes we have proposed the *UML Profile for Collaborative Business Processes based on Interaction Protocols (UP-ColBPIP)* [21, 22, 23]. This is a modeling language defined as UML2 Profile. In addition, in previous works, we have described the generation of technological solutions based on the current different standards by using a MDA approach: ebXML [22], BPEL [24] and WS-CDL [25]. In this way, we had applied a MDA approach to generate B2B specifications and we have also validated the UP-ColBPIP language against

standard languages. The conclusions were most of the elements of the B2B standards can be derived from UP-ColBPIP models.

The purpose of this paper is to describe a complete MDA-based development process for the modeling and specification of the collaborative processes and the partners' interfaces that make up the B2B information system. We present the artifacts (models and code), the languages and techniques to build and transform these artifacts, and the phases and activities of the MDA approach for collaborative processes. We also identify the issues to be considered in each phase of this MDA approach and the patterns of transformations to be applied in order to guarantee the generated B2B specifications fulfill the collaborative processes agreed by the partners in a business level.

This paper is organized as follows. Section 2 describes the phases and the generic pattern of transformations of the MDA approach we propose for collaborative processes. For each phase we describe: the artifacts to be used and built, the techniques to be used and the transformations (mappings) to be defined and applied. Section 3 describes the transformation model tool we have developed to support the transformations required in this MDA approach. Section 4 discusses related work. Section 5 presents conclusions and outlines future research directions.

2 MDA Approach for Collaborative Business Processes

The OMG's MDA Initiative [17] proposes a conceptual framework along with a set of standards (UML, MOF, XMI, etc.) to build model-driven development methods. In MDA, the generic development process consists of: defining platform-independent models (PIMs), selecting platform-specific models (PSMs) and executing transformations that generate PSMs from PIMs, and finally generating Code that can be executed from the PSMs. A platform makes reference to the technology to be used to implement the system. In this way, MDA provides the guidelines and the general artifacts that a model-driven development method has to contain, but it does not provide the concrete development process, the artifacts to be built and the techniques to be used for a particular domain. Furthermore, although UML can be considered as a generic standard modeling language that can also be used to model organizational aspects, it is important to use languages that are more suitable and closer to the application domain.

Therefore, to support the development of collaborative processes and B2B information systems, it is necessary to define a MDA-based development process with the phases, activities, artifacts and techniques required for this application domain.

We propose a MDA approach for collaborative processes in which the development process consists of three phases (Figure 1):
1. Analysis and design of collaborative processes
2. Verification of collaborative processes
3. Implementation of collaborative processes

The artifacts to be produced in the first phase are the technology-independent collaborative process models (Figure 1), which are the inputs for the next phases. In the second phase, the output artifacts are the specifications of collaborative processes in a formal language, in such a way that they can be verified. In this case, according to the

results of the verification, it is possible to go back to the first phase in order to correct the processes. Several cycles of analysis/design and verification can occur. Finally, once the collaborative processes were agreed, the third phase consists of selecting the technology of implementation (i.e. the B2B standard to be used) and generating the B2B specifications that fulfill the collaborative processes defined in the first phase.
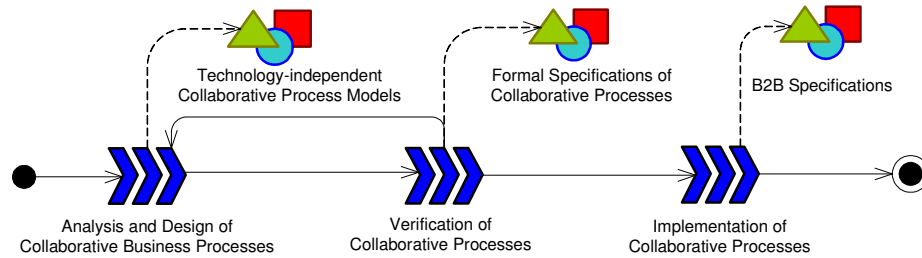

Fig. 1. Phases of the MDA approach for collaborative business processes

To produce the output artifacts of the second and third phases, a generic pattern of transformations is applied according to the principles of MDA, which is shown in Figure 2. This figure indicates the artifacts (components) and the techniques used to build them. The input artifacts are the technology-independent collaborative process models, which are defined in the first phase. The intermediate artifacts are the technology-specific models and the output artifacts are the XML-based specifications.

Two types of transformations have to be defined and executed: transformations of technology-independent models into technology-specific models, and transformations of technology-specific models into XML-based specifications. The first one requires the use of a model transformation tool that enables the definition and execution of these transformations. In addition, it requires the building of the metamodels corresponding to the technology-specific models to be generated. These metamodels can be derived from the XML schemas provided by the technology-specific languages. In this way, the metamodel of a language corresponds to its XML schema; and the models correspond to the XML documents that contain the XML-based specifications. Although the XML documents may be generated directly from the technology-independent models, this intermediate representation allows a more modular and maintainable process transformation and it is in line with the principles of MDA.

The second type of transformations supports the generation of XML documents corresponding to the B2B specifications, according to the technology-specific language selected. These transformations are almost direct through the applications of XML production rules that convert a UML class model into a XML version.

In the next subsections we describe the different phases of the MDA approach for collaborative processes. Subsections 2.2 and 2.3 describe the application of the generic pattern of transformations to implement the transformations required in the second and third phases. Section 3 describes the model transformation tool we have developed and applied to support the definition and execution of these transformations.
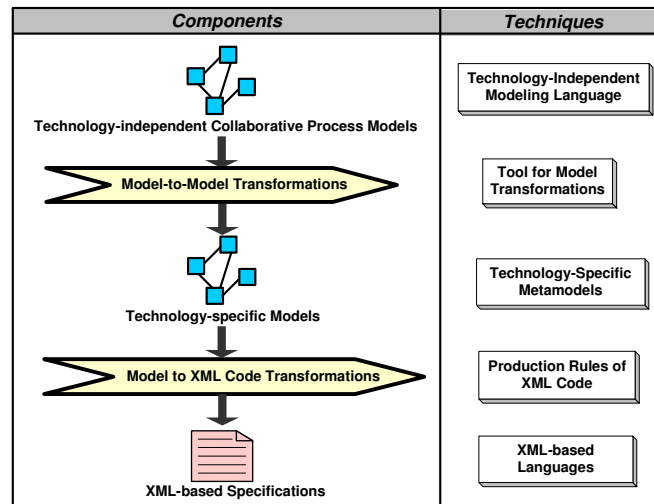
Fig. 2. Generic Pattern of Transformations

2.1. Phase 1: Analysis and Design of Collaborative Processes

This phase is about the modeling of collaborative processes from a business perspective of the B2B collaboration. People involved in this phase are business analysts and system designers, who are the responsible of defining the business aspects of the B2B collaboration. This phase focuses on the analysis and design of the collaborative business processes. The *analysis* stage treats with the identification of the business requirements, i.e. details of the collaboration agreement, common business goals, etc., and the identification of the collaborative processes required to achieve the common business goals agreed by the partners. This stage is defined according with the collaborative business model that partners have agreed for the management of the B2B collaboration. The *design* stage treats with the detailed definition of the behavior of the identified collaborative processes, with the purpose of establishing explicitly the way in that partners will both coordinate their actions and exchange information. In addition, the design also implies the definition of the business interfaces of the partners, which can be derived from the collaborative processes.

The analysis and design of collaborative processes require the building of collaborative process models using concepts that are much less bound to the implementation technology and are much closer to the B2B collaborations domain. This is due to several reasons:

- Business aspects have to be considered in this phase.
- Business aspects are defined by business analysts or system designers who are more interested on the problem domain that in the technical details of implementation.
- People involved in this phase are not acquainted about the different B2B standards and technology-specific languages that can be used to implement the collaboration.

In this way, the analysis and design of collaborative processes imply the modeling of these processes in an independent way of the technology. To support this requirement we have proposed the *UML Profile for Collaborative Business Processes based on Interaction Protocols (UP-ColBPIP)*. This modeling language has been defined as a UML Profile that extends the semantics of UML2 to model technology-independent collaborative processes. This language encourages a top-down approach and provides the conceptual elements to support the modeling of four views (Figure 3):

- *B2B Collaboration View*. This view captures the participants (partners and the roles they fulfill) and their communication relationships in order to provide an overview of the B2B collaboration. UP-ColBPIP extends the semantics of UML2 collaborations to represent B2B collaborations. Furthermore, this view defines the collaborative agreement parameters and the hierarchy of common business goals to be fulfilled by the partners in a B2B collaboration. They are represented extending the semantics of classes and objects of UML.

- *Collaborative Processes View*, which identifies the collaborative processes required to achieve the common business goals. To define this, UP-ColBPIP extends the semantics of use cases to represent collaborative processes as informal specifications of a set of actions performed by trading partners to achieve a common goal.

- *Interaction Protocols View*, which defines the explicit behavior of the collaborative processes through the use of interaction protocols. UP-ColBPIP extends the semantics of the UML2 interactions to model interaction protocols.

- *Business Interfaces View*, which offers a static view of the collaboration through the definition of the business interfaces of the roles performed by the partners. The business interfaces contains the business services (operations) that support the exchange of messages of the interaction protocols. To define this view, UP-ColBPIP extends the semantics of the composite structure and interfaces of UML2.

The first and second views correspond to the analysis stage. The last views correspond to the design stage. The output of this phase is a UP-ColBPIP model with the definition of the above four views, which will contain the description of the collaborative business processes to be carried out in a B2B collaboration.

Due to space limitations, we do not describe how the different views can be defined with the UP-ColBPIP language. More details about this language can be found in [21, 22, 23]. Following, we describe the most important view: Interaction Protocols.

In addition to support a methodological guide for the analysis and design of collaborative processes, the main contribution of this language is the use of interaction protocols to represent the behavior of collaborative processes, along with the application of the speech act theory [18]. In the B2B collaborations domain, an *interaction protocol* describes a high-level communication pattern through a choreography of business messages between partners playing different roles [23].

The purpose of modeling collaborative processes based on interaction protocols is to fulfill the identified requirements for B2B collaborations [22, 23]: enterprise autonomy, decentralization, peer-to-peer interactions, global view of the collaboration and support for negotiations.

In contrast to activity-oriented business process languages, the modeling of interaction protocols focus on the exchange of business messages representing interactions between partners. In this way, interaction protocols support the autonomy required by

the enterprises because their internal activities, services and decisions cannot be defined in interaction protocols and hence, they are kept hidden to its other partners.
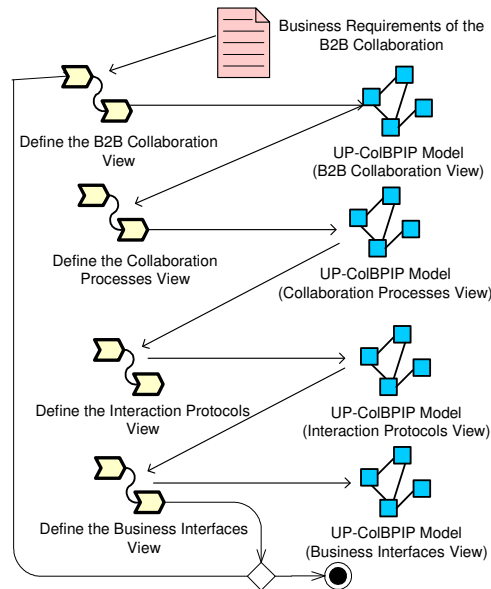

Fig. 3. Analysis and Design Phase of the MDA Approach for Collaborative Processes

Modeling interaction protocols, the focus is on representing the global view of the interactions between the partners. The message choreography describes the global control flow of the peer-to-peer interactions between the partners, as well as the responsibilities of the roles they fulfill. This is the main different with activity-oriented business process languages such as UML Activity Diagrams or BPMN [16], which are more suitable to describe private processes or public (abstract) processes but from the viewpoint of one partner. Although they can also be used to define collaborative processes, to express the global view and the parallel processing of the parties, additional semantics is required for UML2 Activity diagrams and using BPMN, the definition of public processes of each partner is required in order to understand this global view.

In addition, B2B interactions cannot be restricted to mere information transfer [10] but they have to focus on the communicative actions between the parties. These communicative aspects are not also considered in many of the business process modeling languages. In UP-ColBPIP, communicative aspects of B2B interactions are represented in interaction protocols through the use of speech acts. In an interaction protocol, a business message has a speech act associated, which represents the intention that a partner has with respect to an exchanged business document through the message. Thus, decisions and commitments done by the partners can be known from the speech acts. This enables the definition of complex negotiations in collaborative processes.

As an example, we describe the interaction protocol *Demand Forecast Request*, which realizes a simplified process to manage collaborative demand forecasts. Figure

4 shows the sequence diagram of this protocol, in which partner "A" plays the *supplier* role and partner "B" plays the *customer* role. They are represented by lifelines.
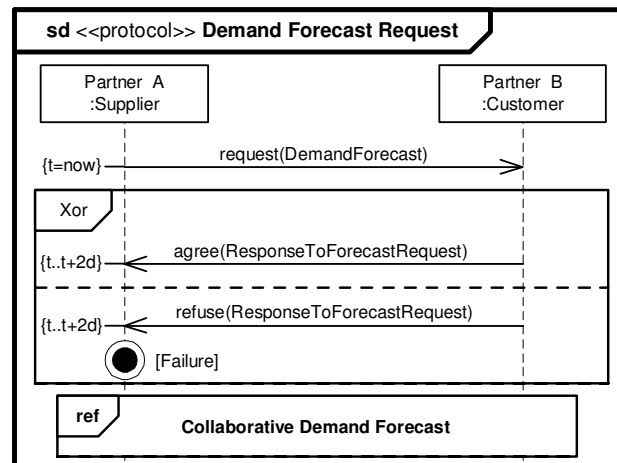

Fig. 4. *Demand Forecast Request* Interaction Protocol

The purpose of this protocol is the supplier and the customer agrees on the type of forecast demand to be defined. The protocol starts with the supplier, which requests a forecast demand, as it is indicated by the message *request(DemandForecast)*. The business document of this message contains the information about the products, periods, and the time horizon of the forecast. Then, the customer processes the received message with its contents and makes a decision. The customer can respond of two different forms, as it is expressed by the control flow segment with the *Xor* operator. This operator indicates that only one of the two paths can occur according to the condition defined in each path. One alternative is the customer accepts the supplier's request and commits to carry out the demand forecast in the future. It is expressed by the message *agree(ResponseToForecastRequest)*. The business document of this message contains more information about the customer's response. Then, the execution of the protocol continues. Another alternative is the customer sends a message *refuse(ResponseToForecastRequest)* to indicate that it rejects the type of forecast requested by the supplier. The business document of this message contains the reasons of the rejection. In this case, the protocol finishes with a failure, as it is indicated by the explicit termination event *failure*. This is only an indication that the protocol does not finish in a correct way from the point of view of the business logic. This failure event does not indicate a technical failure on the communication.

If the customer agrees on carry out the forecast, then the nested protocol *Collaborative Demand Forecast* is invoked in order to partners agree on a common demand forecast. After of the execution of this nested protocol, the protocol finishes. In this case the termination of the protocol is defined in an implicit way.

Furthermore, the business messages of the control flow segment have defined time constraints representing deadlines, which indicate these messages have to be sent before two days, after of the occurrence of the first message.

2.2. Phase 2: Verification of Collaborative Processes

Due to collaborative processes describe the behavior of the partners in the collaboration, as well as the form in which the partners' systems will interact, the verification of functional requirements of these processes, such as the absence of deadlocks and livelooks, is required.

Currently, there are different proposals to verify business processes in technology-specific languages such as BPEL [8] and WS-CDL [9]. Thus, applying a model-driven approach it is possible to generate technological solutions and then verify the generated B2B specifications. However, the main disadvantage of this procedure is that the verification of processes is done in a late stage of the development, when the technological solution has been already defined. Hence, if results of the verification indicate modifications should be done to fix the B2B specifications, these modifications also should be propagated automatically to the collaborative process models. However, these modifications have to be done using the B2B standard languages selected instead of using a technology-independent collaborative process language.

The use of technology-independent models to verify system's properties in an early stage of the development has been recognized as one of main aspects to be supported by a model-driven method [19], because in a early stage of the development is when business analysts and system designers make most of the fundamental decisions.

Therefore, we propose that the verification of the collaborative processes should be done in the second phase of the development process, after of the definition of the technology-independent collaborative process models. This implies that the verification should be applied on these models instead of the B2B specifications.

To achieve this, the generic pattern of transformations we have defined above can be applied for generating the specifications of the collaborative processes in a formal language. In our MDA approach we are using the Petri net formalism to express the semantics of the interaction protocols and enable the verification of their properties.

Figure 5 shows the pattern of transformations to be applied in this phase. The input is the UP-ColBPIP model that contains the defined collaborative processes based on interaction protocols. Model-to-Model Transformations are applied to generate Petri net models. These models are defined according to the language *PNML (Petri Net Marking Language)* [5], which is a XML-based standard language for Petri nets. Hence, the PNML metamodel has to be generated from the XML schema of PNML to enable the building of PNML models.

Finally, from these models, XML documents that contain the Petri nets specifications corresponding to the interaction protocols are generated. In this way, Petri Nets tools can then be used to read the generated Petri nets specifications and to carry out the verification procedures in order to determine if the collaborative processes based on interaction protocols are well-defined and fulfill the desirable properties. The properties to be verified on Petri nets representing protocols are those typical such as

the absence of deadlocks and livelocks. Also, analyzing Petri nets of the protocols enables the detection of structural errors, such as the absence of conditions or deadlines on loop control flow segments or the absence of acknowledgments on messages.
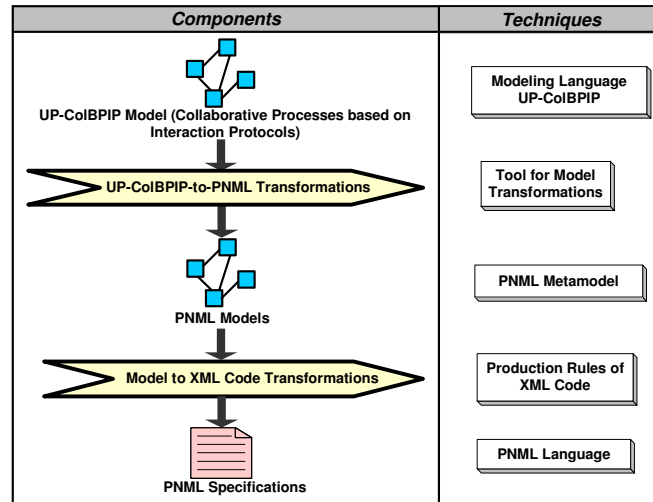

Fig. 5. Pattern of Transformations to generate Petri Nets Specifications

2.3. Phase 3: Implementation of Collaborative Processes

The last phase corresponds to the implementation of collaborative processes, i.e. the generation of specifications of both the processes and the partners' interfaces by using a particular B2B standard. The first activity to be carried out is to select the B2B standards to be used. This has to be agreed by the partners according to their technological requirements.

As we have described above, the generation of B2B specifications requires the use of different languages: one to specify business processes and another one to specify partners' interfaces. In this way, we need to manipulate technology-specific business process models and technology-specific partners' interfaces models.

Figure 6 shows the pattern of transformations to be applied in this phase. The input again is the UP-ColBPIP model that contains the defined collaborative processes based on interaction protocols and the business interfaces of the partners. Two types of models have to be generated from a UP-ColBPIP model: technology-specific business process models and technology-specific partners' interfaces models.

On one hand, the number of technology-specific business process models to be generated depends of the language used. For example, by using ebXML BPSS only one model have to be generated because a XML document of an ebXML BPSS specification can have several binary collaborations representing collaborative processes [22]. This is the same with WS-CDL that allows the definition of several choreo-

graphies in a WS-CDL specification [25]. But it is different with BPEL. By using BPEL, the semantics of a collaborative process based on interaction protocol is represented by means of BPEL abstract processes (also known as public processes), one by each partner involved in the protocol [24]. This means that a BPEL abstract process represents the behavior of the role fulfilled by one partner in a collaborative process. Hence, the number of BPEL abstract processes models to be generated from a UP-ColBPIP model is the sum of all of the roles involved in all the interaction protocols.
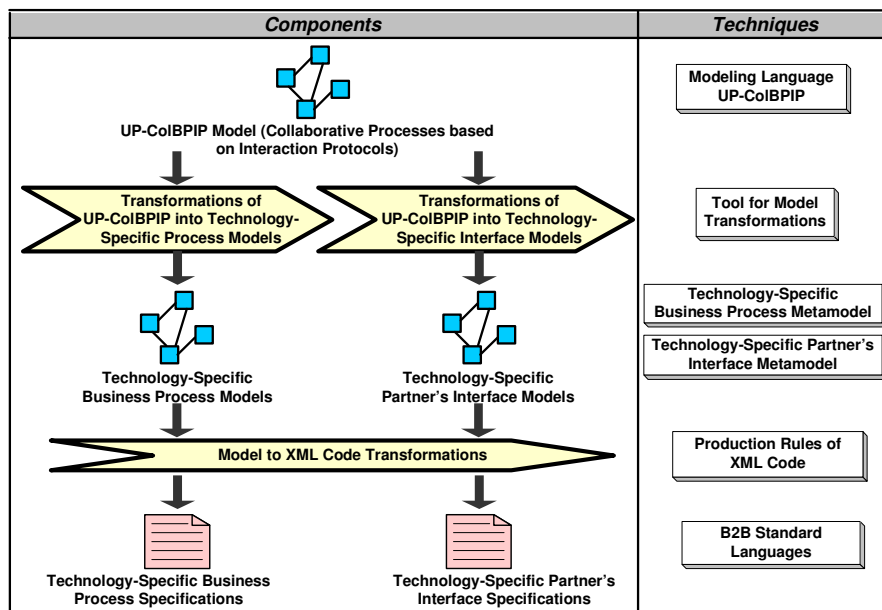

Fig. 6. Pattern of Transformations to generate B2B Specifications

On the other hand, the number of technology-specific partners' interfaces models depends of the number of partners defined in the UP-ColBPIP model. For each partner, a technology-specific partner's interface model is generated. To define partners' interfaces, ebXML provides the CPPA (Collaboration-Protocol Profile and Agreement) language, and standards based on web services composition depend on the Web Services Description Language (WSDL).

Finally, from these technology-specific models, the XML-based specifications of business processes and partners' interfaces are generated.

3 Tool for Model Transformations

The transformation of UP-ColBPIP models into formal or B2B specifications can be implemented by using the model transformation tool prototype proposed in [21]. This tool enables the definition of model transformations for generating specifications based on a XML-language from UP-ColBPIP models.

The architecture of this tool is shown in Figure 7. The Transformation Editor (TE) implements a model transformation language that supports the declarative definition of transformation rules and their composition in rule modules consisting of a hierarchy of rules. Model transformations represent the transformations of UP-ColBPIP models into technology-specific models. A model transformation contains a set of transformation rules and rule modules. For each mapping between elements of the source and target languages, a transformation rule is defined. Transformation rules consist of LHS (Left-Hand Side) and RHS (Right-Hand Side) patterns defined according to the metamodels of the source language and the target language. These patterns are specified in an imperative way using Java code. An API is provided to define these patterns. The execution order of the rules is defined in the rule modules. A rule defined in a rule module can call to another module in order to allow recursive rules. Model Transformations are saved in a XML format. The tool contains a repository for the storage and retrieval of model transformations for the subsequent execution.
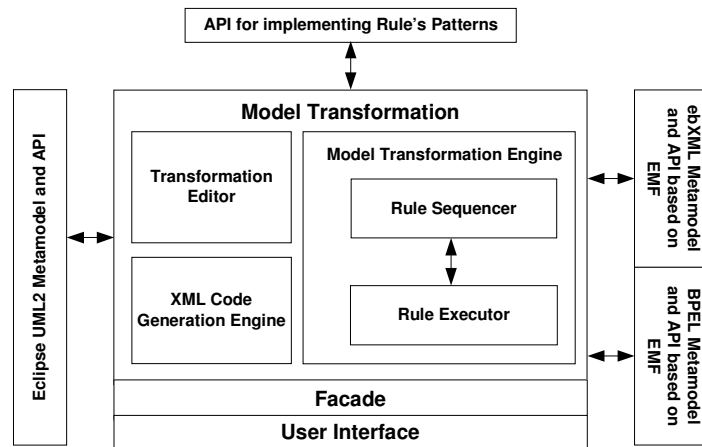

Fig. 7. Architecture of the Model Transformation Tool.

To support the execution of model transformations and generate the XML code of the specifications, the tool uses the Transformation Model Engine that takes as input a UP-ColBPIP model in an XMI format and generates technology-specific models according to the technology-specific language used. To do this, the transformation model engine has two subcomponents: the Rule Sequencer, which interprets the rule models to determine the order of execution of the rules; and the Rule Executor, which interprets each transformation rule and executes its corresponding LHS and RHS patterns. To generate XML documents (corresponding to the specifications) from the above models, the tool has a XML Code Generation Engine, which implements production rules of XML code from object models.

To manipulate UP-ColBPIP models and import them in a XMI format, the tool uses the APIs of the Eclipse UML 2 plug-in [7], which is an implementation of the UML 2 metamodel based on the Eclipse Modeling Framework (EMF) [6]. Hence, UP-ColBPIP models can be defined with any UML Case tool that implements the Eclipse

UML2 Metamodel. Also, for manipulating technology-specific models, EMF was also used to generate the metamodel and APIs that allow the manipulation of these models.

The Tool has a 3-tier architecture. The Facade separates the user interfaces tier from the main tier that contains the components described above. In this way, the user interfaces can be implemented to be used from HTML browsers or traditional clients. Currently, we have implemented this last type of client.

4 Related Work

The application of model-driven development (MDD) approaches has been exploited in the domain of web services to generate technological solutions based on web services composition [2, 15]. However, as we have appointed in previous works [24, 25], they support the modeling and specification of the behavior of collaborative processes but only from the point of view of one partner. As a result, they do not support a global view in the modeling of collaborative processes. From the methodological point of view these approaches do not also define a complete development process. In addition, they are not based on a standard conceptual framework such as MDA.

In [13], a complete development process for the generation of web services compositions was defined. However, the principles of MDA are only applied in the phase of generation of the process definitions. The verification of technology-independent business process models is not considered.

The above mentioned MDD approaches are only focused on the domain of web services and therefore, they do not also consider the application of a technology-independent modeling language for collaborative processes.

In [11,12], the transformation of technology-independent collaborative processes defined with the UN/CEFACT Modeling Methodology (UMM) to B2B specifications has been proposed. However, UMM do not provide a complete development process to generate B2B specifications. It only provides a development process for modeling technology-independent collaborative processes. In addition, the proposal defined in [11, 12] does not consider neither the application of a model-driven approach nor the verification of the processes in an early stage of the development.

Different to the above works mentioned, the development process defined in this paper exploits the principles of MDA in each phase of the development, through the application of different patterns of transformations according with the type of XML-based specification to be generated.

5 Conclusions and Future Work

In this work, we have described the development process of a MDA approach we propose for the modeling and specification of collaborative business processes. As part of this MDA approach, we have described the phases and activities to be carried out, the artifacts to be produced and the techniques to be used for building these arti-

facts. Through this MDA approach we contribute with a model-driven development method for enabling enterprises to establish B2B collaborations with their partners.

First, as a start point of the development, we have highlighted the importance of using a technology-independent modeling language for the analysis and design of collaborative processes. Hence, we propose the use of the UP-ColBPIP language. This language supports a top-down approach and encourages the use of interaction protocols for representing the behavior of collaborative processes. The use of interaction protocols allows fulfilling with the requirements of the B2B collaborations.

Second, the verification of collaborative processes previous to the generation of the technological solutions is very important in order to determine, in an early stage of the development, if collaborative processes defined in a business level are well-formed. To do this, we have indicated the pattern of transformations required to generate Petri Nets specifications based on the PNML language from UP-ColBPIP models. PNML-based specifications can be used by most of the Petri Nets Tools to verify properties of the Petri Nets representing the interaction protocols.

Third, we have indicated the pattern of transformations required to generate B2B specifications that allow the implementation of collaborative processes. Two types of technology-specific models and specifications have to be generated: those representing business processes and those representing interfaces of the partners. The number of models and specifications to be generated depends on the B2B standard selected.

Finally, we have described a model transformation tool developed to generate B2B specifications. This tool implements a particular model transformation language that allows the definition of rules that can be reused in different rule modules to specify the ordering, composition and recursive definition of the transformation rules. The patterns of the rules are defined in an imperative way. The advantage of using this tool is it was developed specifically for the application domain of collaborative processes. However, the main disadvantage of this tool is that patterns of the rules are defined in an imperative way, and hence, it is necessary developers with a well-known of the metamodels of the languages and the Java APIs that allow the manipulation of the models. Furthermore, correctness in the definition of the transformations cannot also be guaranteed. To overcome these disadvantages, we are extending this model transformation tool to define and execute rules based on graph transformations. The purpose is to reuse as model transformation engine a tool that provides a graph transformation engine for the execution of transformations and enables the graphical definition of the patterns of the rules as graphs.

Our ongoing work is also about the building of a software development environment to support the management of this complete MDA-based development process.

References

1. Baghdadi, Y.: ABBA: An architecture for deploying business-to-business electronic commerce applications. Electronic Commerce Research and Applications, 3(2004) 190-212
2. Baïna, K, Benatallah, B., Cassati, F., Toumani, F.: Model-Driven Web Service Development. CaiSE'04, Springer (2004) 290-306.

3. Bernauer, M., Kappel, G., Kramler, G.: Comparing WSDL-based and ebXML-based Approaches for B2B Protocol Specification. Proceedings of ISOC'03 (2003).

4. BEA, IBM, Microsoft, SAP, Siebel: Business Process Execution Language for Web Services. http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003.

5. Billington, J., Christensen, S., van Hee, K.E., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C.: The Petri Net Markup Language: Concepts, Technology, and Tools. 24th Int. Conf. on Applications and Theory of Petri Nets, LNCS 2679 (2003) 483-505

6. Eclipse: Eclipse Modeling Framework. http://www.eclipse.org/emf/

7. Eclipse: Eclipse UML2 Project. http://www.eclipse.org/uml2/

8. Ferrara, A: Web Services: a Process Algebra Approach. The Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004). ACM (2004)

9. Foster, H., Uchitel, S., Magee, J., Kramer, J. Model-Based Analysis of Obligations in Web Service Choreography. IEEE International Conference on Internet&Web Applications and Services, Guadeloupe, French Caribbean (2006).

10. Goldkuhl, G., Lind,M.: Developing E-Interactions – a Framework for Business Capabilities and Exchanges. Proceedings of the ECIS-2004, Finland, 2004.

11. Hofreiter, B., Huemer, C.: Transforming UMM Business Collaboration Models to BPEL. International Workshop on Modeling Inter-Organizational Systems (MIOS), 2004.

12. Hofreiter B., Huemer C.: ebXML Business Processes - Defined both in UMM and BPSS. Proc. of the 1st GI-Workshop XML Interchange Formats for Business Process Management, Modellierung 2004, Germany, 81-102, 2004.

13. Karastoyanova, D.: A Methodology for Development of Web Service-based Business Processes. Proceedings of the First Australian Workshop on Engineering Service-Oriented Systems (AWESOS 2004), ASWEC 2004, Melbourne, Australia (2004)

14. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y.: Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation (2005).

15. Koehler, J., Hauser, R., Kaporr, S., Wu, F., Kurmaran, S.: A Model-Driven Transformation Method. 7th International Enterprise Distributed Object Computing, 2003.

16. OMG/BPMI. Business Process Modeling Notation, Version 1.0, 2004.

17. Object Management Group: MDA Guide V1.0.1, 2003. http://www.omg.org/mda.

18. Searle, J.R.: Speech Acts, an Essay in the Philosophy of Language, Cambridge, 1969.

19. Selic,B :The Pragmatics of Model-Driven Development. IEEE Software,20(5),19-25(2003)

20. UN/CEFACT and OASIS: ebXML Business Specification Schema Version 1.10, http://www.untmg.org/downloads/General/approved/ebBPSS-v1pt10.zip (2001)

21. Villarreal, P.: Method for the Modeling and Specification of Collaborative Business Processes. PhD Thesis. Universidad Tecnológica Nacional, Santa Fe, Argentina (2005).

22. Villarreal, P., Salomone, H.E. and Chiotti, O.: Applying Model-Driven Development to Collaborative Business Processes. Proceedings of the 8th Ibero-American Workshop of Requirements Engineering and Software Environments (IDEAS 2005), Chile, 2005.

23. Villarreal, P., Salomone, H.E. and Chiotti, O.: Modeling and Specifications of Collaborative Business Processes using a MDA Approach and a UML Profile. In: Rittgen, P. (eds): Enterprise Modeling and Computing with UML. Idea Group Inc, (in press).

24. Villarreal, Salomone, Chiotti: MDA Approach for Collaborative Business Processes: Generating Technological Solutions based on Web Services Composition. 9th Ibero-American Workshop of Requirements Engineering and Software Environments, Argentine, (2006).

25. Villarreal, P., Salomone, H.E. and Chiotti, O.: Transforming Collaborative Business Process Models into Web Services Choreography Specifications. 2dn IEEE International Workshop on E-Commerce and Services, LNCS 4055 (2006) 50-65 (in press)

Deriving a textual notation from a metamodel:
an experience on bridging *Modelware* and *Grammarware*

Angelo Gargantini *, Elvinia Riccobene **, and Patrizia Scandurra ***

Abstract In this paper, we show how the OMG's metamodelling approach to *domain-specific language* definition can be exploited to infer human-usable textual notations (concrete syntaxes) from the conceptualization provided by metamodels (abstract syntaxes). We give general rules to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we show how to instruct a *parser generator* by these rules for generating a *compiler* which is able to parse the grammar and to transfer information about models into a MOF-based instance repository. The approach is exemplified for the *Abstract State Machines Metamodel* (AsmM), a metamodel for the Abstract State Machine (ASM) formal method, by illustrating the derivation of a textual notation to write ASM specifications conforming to the AsmM, and the process to input ASM models into a MOF repository.

1 Introduction

The Model-driven Engineering (MDE) [8] conceives *metamodelling* as a modular and layered way to endow a language or formalism with an *abstract notation*, so separating the abstract syntax and semantics of the language constructs from their different *concrete notations*. A language has to be equipped by at least a proper MOF-compliant metamodel which provides the language abstract syntax, an easy to learn concrete syntax, possibly graphic, a well-founded semantics, and a uniform style (through, e.g., the XML base format [29]) of representing language constructs for interchanging purposes.

The process of deriving textual or graphical concrete notations from a MOF compliant metamodel (*forward engineering*) is not yet well established, while the opposite, from EBNF grammars to MOF (*reverse engineering*), has been more intensively studied [14,17,6,27]. The forward process is more demanding than the reverse, since MOF-based metamodels inherently contain more information than EBNF grammars. Metamodels are graphs with special edges that specify different nodes relationships (generalizations, aggregations, compositions, and so on), whereas EBNF grammars can be presented as tree of nodes and directed edges, but the edges do not contain as much information as properties in a metamodel. A mapping from EBNF grammars to metamodels uses only a subset of the capabilities of metamodels, and the generated metamodel may need to be further enriched in order to make it more abstract.

Complex MOF-to-text tools, capable of generating text grammars from specific MOF-based repositories, exist [16,10]. These tools render the content of a MOF-based

---

* University of Bergamo, Italy - email: angelo.gargantini@unibg.it
** University of Milan, Italy - email: riccobene@dti.unimi.it
*** University of Catania, Italy - email: scandurra@dmi.unict.it

repository (known as a *MOFlet*) in textual form, conforming to some syntactic rules (grammar). However, although automatic, since they are designed to work with any MOF model and generate their target grammar based on predefined patterns, they do not permit a detailed customization of the generated language.

The work presented in this paper is part of our effort toward the development of a general framework for Abstract State Machine (ASM) tools interoperability. Indeed, the increasing application of the ASMs as formal and engineering method for hardware and software systems development, has caused a rapid development of tools around ASMs of various complexity and goals: tools for mechanically verifying properties using theorem provers or model checkers and execution engines for simulation and testing purposes [9]. However, ASM tools have been usually developed by individual research groups, are loosely coupled and have syntaxes strictly depending on the implementation environment. This makes the integration of tools hard to accomplish and prevents ASMs from being used in an efficient and tool supported manner during the software development life-cycle.

In [22,15], we adopted the MDE suggestion upon which a wide interoperability of tools can be reached through metamodels and model transformations: the metamodel of each tool is linked to those of other tools by a logical *pivot metamodel* which abstracts a certain number of general concepts and constructs about the domain-specific language. We defined a metamodel for ASMs with the intention to use it as the pivot metamodel toward a definition of a standard family of languages for the ASMs and a systematic integration of a number of ASM tools upon metamodelling techniques.

After developing the pivot metamodel, as further step, we intended to define a concrete syntax derived from the metamodel as textual notation to write ASM models conforming to the metamodel. Since ASM is not an object-oriented formalism (even if it can model OO concepts), we did not want to use existing tools as HUTN or Anti-Yacc since they provide concrete notations strongly reflecting the object-oriented nature of the MOF meta-language. Therefore, we define general rules how to derive a context-free EBNF (Extended Backus-Naur Form) grammar from a MOF-compliant metamodel, and we show how to use these mapping rules to instruct the JavaCC *parser generator* for generating a *compiler* which is able to parse the grammar and to transfer information about models into a MOF-based instance repository.

Although the task (possibly iterative) of defining a metamodel for a language is not trivial and its complexity closely matches that of the language being considered, we like to remark that the effort of developing from scratch a new EBNF grammar for a complex formalism, like the ASMs, would not be less than the effort of realizing a MOF-compliant metamodel for the ASMs, and then deriving a EBNF grammar from it. Moreover, the metamodel-based approach has the advantage of being suitable to derive from the same metamodel (through mappings or projections) different alternative concrete notations, for various scopes like graphical rendering, model interchange, standard encoding in programming languages, and so on.

The paper is organized as follows. ASMs and the ASM metamodel are presented in Section 2. Section 3 introduces the rules for deriving an EBNF grammar from a MOF metamodel and discusses how to instruct the JavaCC parse generator to build our

compiler. An example of ASM written in the textual notation derived by the metamodel is shown in Section 4. Related and future work are given in Sections 5 and 6.

2   Overview of the case study

Abstract State Machines *Abstract State Machines* (ASMs) [9] are a system engineering method that guides the development of hardware and software systems seamlessly from requirements capture to their implementation. Even if the ASM method comes with a rigorous scientific foundation [9], the practitioner needs no special training to use it since Abstract State Machines are a simple extension of Finite State Machines, and can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures. The computation of a machine is determined by firing *transition rules* describing the modification of the functions from one state to the next. The notion of ASMs moves from a definition which formalizes simultaneous parallel actions of a single agent, either in an atomic way, *Basic ASMs*, and in a structured and recursive way, *Structured or Turbo ASMs*, to a generalization where multiple agents interact in a synchronous way, or asynchronous agents proceed in parallel at their own speed and whose communications may provide the only logical ordering between their actions, *Synchronous/Asynchronous Multi-agent ASMs*.

We assume the reader is familiar with the ASM method. A complete presentation on the ASMs and their successful application in different fields can be found in [9].

The AsmM metamodel The *Abstract State Machines Metamodel* (AsmM) is a complete meta-level representation of ASMs concepts based on the OMG's MOF metalanguage. The specification of AsmM [7] comprises: (i) an *abstract syntax*, i.e. a MOF-compliant metamodel and OCL constraints representing in an abstract (and visual) way concepts and constructs of the ASM formalism as described in [9]; (ii) an *interchange syntax*, i.e. a standard XMI-based format automatically derived from the AsmM, for the interchange of ASM models; and, (iii) a Java application program interface (API) based on the standard Java Metadata Interfaces (JMI) [18] for MOF 1.4 for managing and accessing metadata (that in our case are ASM models) in a MOF-based repository. For the metamodel *semantics*, we adopt the ASM semantics in [9].

We developed the metamodel in a modular way reflecting the natural classification of ASMs in Basic, Turbo, and Multi-Agent (Sync/Async). Metamodelling representation results into class diagrams (a natural visual rendering of MOF models). Each class is also equipped with a set of relevant *constraints*, OCL invariants written to fix how to meaningful connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams. The complete metamodel contains about 115 classes, 114 associations, and 130 OCL constraints and is organized in one package called `ASM` which is further divided into five sub-packages:

- the ASMStructure package or *the structural language* defines the architectural constructs (modules and machines) required to specify the backbone of an ASM model (this package contains the hierarchy of classes rooted by the class `Asm`);
- the ASMDefinitions package or *the definitional language* contains constructs (functions, domains, rule declarations, etc.) which characterize algebraic specifications;

– the ASMTerms package or *the language of terms* provides all kinds of syntactic expressions which can be evaluated in a state of an ASM;
– the ASMTransitionRules package or *the language of rules* contains all possible transition rules scheme of Basic and Turbo ASMs;
– the DataTypes package specifies further MOF types used to define the AsmM itself (for typing classes attributes).

A standard AsmM library of predefined domains and functions is provided in [7].

We have drawn the AsmM with the Poseidon UML tool (v.4.0) [21] empowered with an ancillary tool UML2MOF which transforms UML models to MOF 1.4 and is provided by the MDR Netbeans framework [3]. The XMI format has been generated automatically from the AsmM: according to the MOF 1.4 to XMI 1.2 mapping [28], a XML document type definition file, commonly named DTD, has been generated from the AsmM in the MDR framework. Similarly, the AsmM JMI was obtained automatically by the MDR framework according to the standard MOF-to-Java mapping [18]. Through the AsmM JMI, a Java client (like the parser generated by the JavaCC, see sec. 3) can access the AsmM packages, create instances of classes of the AsmM, and set and modify attributes and associations of these instances.

3 From MOF to EBNF: how to generate a context-free grammar and instruct a parser generator

In this section, we give mapping rules to derive an EBNF grammar from a MOF metamodel. Even if they have been used to provide the ASM formal method with a textual notation, here called *AsmM-CS*, conforming to the AsmM, they are general enough and do not rely on the specific domain language.

For the MOF to EBNF mapping, we take into account all MOF 1.4 constructs which bring information about the domain knowledge, except constructs like operations and exceptions which are related to the execution semantics of MOF-based repositories rather than to the concepts being meta-modeled, and packages which are used to group elements within a metamodel for partitioning and modularization purposes only. These constructs, however, and, in general, the whole structure of the metamodel are taken into account inside the parser to instantiate and query the content of a MOF repository.

We also provide guidance on how to assemble a JavaCC file given in input to the JavaCC [1] parser generator to automatically produce a parser for the EBNF grammar of the AsmM-CS. This parser is more than a grammar checker: it is able to process ASM models written in AsmM-CS and to create instances of the AsmM in a MDR MOF repository trough the use of the AsmM JMIs.

A JavaCC file contains a sequence of Java-like method declarations each representing the EBNF production rule for a non terminal symbol and corresponding to an identically named method in the final generated Java parser. Each JavaCC method begins with a set of Java declarations and code (to access the MOF repository, create instances of the classes of the metamodel using the AsmM JMIs), which become the initial declarations and code of the generated Java method and hence are executed every time the non-terminal is parsed.

A JavaCC method continues with the *expansion unit* statement, or parser actions, to instruct the generated parser on how to parse symbols and make choices. The expansion unit corresponds to the `<expression with symbols>`of the EBNF rule and may contain Java code within braces to perform actions like set attributes and references. The expansion unit can also include *lookaheads* of various types – local, syntactic, and semantic – (see [1] for details). Lexical and syntactical analysis errors can be caught and reported using standard Java exception handling. The JavaCC grammar file for the Asm-CS can be found in [7] and consists of about 6852 lines of code. We report here some fragments of it in `typewriter` font.

Note that, the grammar and the input file for the parser generator can be further optimized and enriched. For example, suitable methods were added to the AsmM-CS in order to allow alternative representations of the same concepts (i.e. a class instance in the metamodel can admit many equivalent notations) such as the interval notation for sets/sequences/bags of reals, special expressions to support the infix notation for some functions on basic domains (like plus, minus, mult, etc.), and so on.

Following sections explain the set of **rules** on how to map MOF 1.4 constructs into EBNF and into JavaCC. We assume the reader familiar with MOF and EBNF.

3.1 Class

A MOF class acts as the namespace for attributes and outgoing role names on associations.

Rule 1: A class *C* is always mapped to a non terminal symbol C. User-defined keywords – optional and chosen depending on how one wants the target textual notation appears – delimit the expression with symbols in the derivation rule for C. The expression represents the actual content of the class and is determined by the *full descriptor*[1] of the class according to the other rules below. For each class *C*, we introduce in JavaCC one method which has the following schema.

```
C C(): { // create result, a new instance of C in the repository
        // temp variables for attributes and references
}{ // expansion unit
   <startC> // expression starting delimiter
   // read content of C and fill the instance result
   <endC> // expression ending delimiter
{ return  result;}}
```

The method has signature `C()` and returns a JMI instance of the class C. When executed to parse the grammar symbol C, it creates a new instance of C called `result` and initializes a list of variables to store attributes and references of *C*. Then it starts parsing the content of *C* enclosed between the keywords `<startC>` and `<endC>`, i.e. it reads attributes and references of *C*, as explained in the following sections, and sets the attributes and references of `result`. In the end it returns `result`.

---

[1] A full descriptor is the full description needed to describe an object. It contains a description of all of the attributes, associations, etc. that the object contains, including features inherited from ancestor classes.
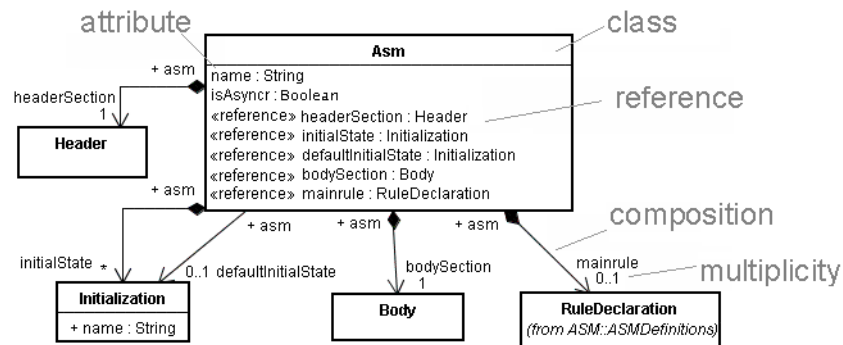
Figure 1. Example of class with attributes, references and associations

Rule 2: The start symbol of the grammar is the non-terminal symbol corresponding to the *root class* of the metamodel, i.e. the class from which all the other elements of the metamodel can be reached.

Example Fig. 1 shows the MOF model of an ASM defined by a name, a Header (to establish the signature), a Body (to define domains, functions, and rules), a mainrule, an Initialization (for the set of initial states), and one initial state elected as default (designed by the association end defaultInitialState). For this class we introduce a non terminal Asm in the grammar and a method `Asm Asm()` in JavaCC. The derivation rule of the non terminal Asm has the keyword "asm" as starting delimiter and no ending delimiter (`<EOF>` in JavaCC code). The class Asm is the root element of the metamodel, therefore its corresponding non terminal is chosen as start symbol of the grammar.

3.2 Multiplicity

Rule 3: Multiplicity values are mapped to repetition ranges. A 0..1 multiplicity (zero or one) is mapped to brackets [ ] or a question mark ?. A * multiplicity (zero or more) corresponds to the application of the Kleene star operator *. A 1..* multiplicity (one or more) corresponds to the the Kleene cross operator +. A n multiplicity (exactly n) corresponds to the operator {n}. A n..* (n or more) multiplicity corresponds to the operator {n,}, a n..m multiplicity (at least n but not more than m) corresponds to the operator {n,m}.

3.3 Data Type

MOF supports two kinds of data type: *primitive data types* like Boolean, Integer, and String; *constructors* like enumeration types, structure types, collection types, and alias types to define more complex types. Primitive data types do not have a direct representation in terms of EBNF elements, while in JavaCC are mapped to the correspondent primitive data types. However, they are used to transform attributes in EBNF concepts (see the next section for details). For structured data types, we do not introduce new

EBNF rules, since each attribute of structured type can be turned in an attribute of Class type by replacing the structured data type with a class definition in the metamodel.

3.4 Attribute (instance-level)

The representation of attributes of a class *C* within the expression on the right-hand of the derivation rule of the nonterminal C depends on the *type* (a MOF data type or a class of the metamodel) of the attribute and on its *multiplicity* (optional, single-valued, or multi-valued). Usually, optional attributes are represented when their value is present, and are not represented when their value is absent.

Rule 4: Attributes of *Boolean* type are represented as keywords (terminal symbols) reflecting the name of the attribute and followed by a question mark ? to indicate it is optional. At instance level, the presence of the keyword in a textual specification indicates that the attribute value is true, and vice-versa.

Rule 5: Attributes of *String* type are represented by a string literal value <STRING> preceded by an optional keyword which reflects the name of the attribute. If a class has an attribute "*name*" of String type, then that attribute is used as *identifier* for objects of the class[2]. We represent the identifier for a class *C* in EBNF by a non terminal <ID_C> which is a sequence of string literals (optional constraints can be given on characters in an identifier). The identifier can be used to retrieve an instance of *C* when needed. In the following, we refer to an object *by name*, if we use its name to univocally refer to it. In JavaCC we introduce a function C getCByName() which reads the string ID_C and retrieves the instance of *C* with that name.

No restrictions are placed on the order of the attribute and reference representations (see sec. 3.6) within the production for the non terminal of a class. Although it is expected that the produced grammar has a consistent ordering of the syntactic parts, such ordering is fixed during the derivation process of the grammar from the metamodel (e.g. through interactive wizards), but no extensions (like stereotypes or special tags) are imposed on the MOF-based metamodel in order to reflect the linear order of EBNF.

Example 1 For the attributes of the Asm class in Fig. 1, by **rule 4** and **rule 5**, we introduce the following EBNF derivation rule and JavaCC method:
Asm ::= "asm" ("isAsyncr")? <ID_Asm>

```
   Asm Asm(): {Asm result = AsmStructure.getAsm().createAsm();
               String name;
               boolean isAsyncr = false;
} { "asm" ["asyncr" { result.setAsynchr(true);}]
               name = <ID_Asm> { result.setName(name);}
               //read the header, initial states, body, ...
      <EOF>
{ return result;}}
```

Rule 6: Attributes of *Enumeration* type are represented as a choice group of keywords which reflect the name of the enum literals.

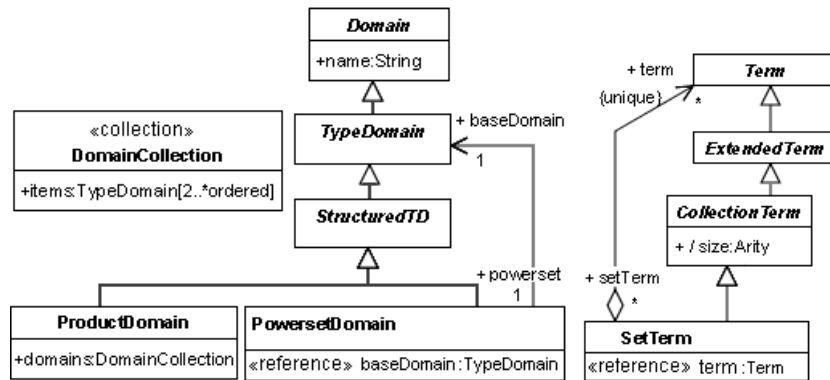[2] Note that in MOF2 the 'isId' attribute can be used for this purpose.

Figure 2. Example of a collection data type and of a multi-valued reference

Rule 7: Attributes of *Integer* type are represented by an optional keyword for the name of the attribute followed by a literal representation of the attribute value.

Rule 8: Attributes of *Collection data type* are represented by an optional keyword which reflect the name of the attribute, followed by a representation of the elements of the collection. Each element can be represented either *by-value*, i.e. by an occurrence of the non terminal of the typing class, or *by-name*, i.e. an occurrence of the identifier if any. Moreover, elements of the collection can be optionally enclosed within parentheses (and), and separated by comma.

Example 2 The class ProductDomain in Fig. 2 has an attribute which is a collection of type-domains. By **rule 1**, we use the initial keyword Prod as delimiter for ProductDomain. We apply **rule 8** omitting the keyword for the attribute domains and we choose to represent the elements of the collection in a by-name fashion (TypeDomain inherits the attribute name from the ancestor class Domain), separated by comma. The feature ordered is reflected by the ordering in the EBNF. The multiplicity 2..* corresponds to the operator {2,}, which is turned into the form $a(a)^+$ with a a syntactic part.

ProductDomain ::= "Prod" "("<ID_Domain> ("," <ID_Domain>)+ ")"

The following method in JavaCC is associated to the class ProductDomain. It creates a new `ProductDomain`, read the delimiters in keyword form and read the list of type-domains by the method `getTypeDomainByName()`, adding them to a new list `domains`. In the end, `domains` is assigned to the domains attribute.

```
    ProductDomain ProductDomain(): { ProductDomain result =
             AsmDefinitions.getProductDomain().createProductDomain();
             Collection domains = new LinkedList();
             TypeDomain td;
}{"Prod" "("  td = getTypeDomainByName()
               {/* add td to the domain list */ domains.add(td);}
          ("," td = getTypeDomainByName()
               {/* add td to the domain list*/
domains.add(td); } )+ ")"
```

```
              { /* set the domains */ result.setDomains(domains);}
  { return result;}}
```

Rule 9: Attributes whose type is a *class* of the metamodel are represented by keywords which reflect the name of the attribute, followed by either a full representation of the instance (or *by-value*), i.e. an occurrence of the non terminal of the typing class, or *by-name*, taking into account the multiplicity.

Rule 10: Attributes of *Alias type* are represented depending on the aliased type.

Rule 11: *Derived attributes*[3] are not mapped to EBNF concepts, since the parser can infer them, and then instantiate them in a MOFlet, from other existing elements (which are instead expressed at EBNF level). In JavaCC they are set at the end of the method, just before returning the result.

Rule 12: Other MOF features like visibility, isLeaf, isRoot, changeability, and default values are not considered for an EBNF representation.

3.5 Association and Association End

Associations are represented in terms of their ends, and association ends are represented in EBNF in terms of their corresponding references (see next section). Only *eligible* association ends are represented (**Rule 13**). An association end is considered eligible[4] if it is navigable, if there is no explicit MOF reference for that end within the same outermost package, and if the association of the end is owned by the same package that owns the type of its opposite end (to avoid circular package dependencies). Moreover, similarly to attributes, derived association ends (even if eligible) are ignored.

3.6 Reference

MOF references are a means for classes to be aware of class instances that play a part in an association, by providing a view into the association as it pertains to the observing instance. Here, MOF references are inferred by each *eligible* association end. Therefore, the EBNF representation of a reference depends on the nature (simple, shared aggregation, composite aggregation) of the association to which it refers.

Rule 14: A reference in a *simple association* (that is, the associated instance can exist outside the scope of the other instance) is represented by an optional keyword, which reflects the name of the reference or the role name of the association end, followed by either a by-value or a by-name representation if any, taking into account the multiplicity. Moreover, referenced collections can be optionally enclosed within parentheses, and the syntactic parts for its elements are separated by comma.

---

[3] The MOF flag *isDerived* determines whether the contents of the notional value holder is part of the explicit state of a class instance, or is derived from other state. Derived attributes or association ends are denoted with a slash (/) preceding the name.

[4] We take this definition from the UML profile for MOF 1.4 by the MDR framework [3]. It is used to automatically imply MOF references by association ends. MOF references are implied, in fact, by each *eligible* UML association end.

Example 1 In Fig. 2, MOF references are shown as attributes with «reference» stereo-type, as implied by each *eligible* association end. By **rule 1** for the class PowersetDomain we decided to use the initial keyword Powerset as delimiter, while by **rule 14** we omit the keyword for the reference baseDomain. This last is represented by-name, and the elements of the referenced collection (in this case just one element) are enclosed within parentheses (and).

EBNF: PowersetDomain ::= "Powerset" "(" <ID_TypeDomain> ")"

In JavaCC we introduce a new method `PowersetDomain`, which reads the delimiters in keyword form and read the type-domain by name calling the method `TypeDomain getTypeDomainByName()`.

```
PowersetDomain PowersetDomain(): { PowersetDomain result =
                AsmDefinitions.getPowersetDomain().createPowersetDomain();
                TypeDomain baseDomain;
}{ "Powerset" "("  baseDomain = getTypeDomainByName() ")"
        { /*set the baseDomain */
result.setBaseDomain(baseDomain);}
{ return result;}}
```

Rule 15: In a *shared aggregation* (white-diamond, weak ownership, i.e. the part may be included in several aggregates) or in a *composite aggregation* (black-diamond, the contained instance does not exist outside the scope of the whole instance), the reference to the contained instance is represented in the production rule of the non terminal corresponding to the whole class in a *by-value* fashion, i.e. as a non terminal (corresponding to the class of the contained instance) preceded by an optional keyword [5] reflecting the name of the reference or of the role end, and combined with other parts of the production taking into account the multiplicity. Moreover, referenced collections can be optionally enclosed within parentheses, and the syntactic parts for its elements are separated by comma. A reference to the whole instance (if any) is not represented.

Example 2 By **rule 1** for the class SetTerm in Fig. 2 we decided to use the keywords { and } as delimiters, while by **rule 15** we omit the keyword for the reference term. This last is represented in a by-value fashion, and the elements of the referenced collection are not enclosed within parentheses, but are separated by comma. Finally, by **rule 11** the derived attribute size is not represented at EBNF level; however (see the JavaCC code below) inside the parser its value is calculated and set accordingly.

EBNF: SetTerm ::= "{" Term ("," Term)* "}"

The term reference in the SetTerm class is a multi-valued reference. In this case the reference is set by adding elements to the collection returned by the JMI operation `public java.util.Collection getTerm()`. This JMI method returns the value of the reference term, i.e. the collection of elements (as terms) of the set-term.

```
SetTerm SetTerm():{
SetTerm result = AsmTerms.getSetTerm().createSetTerm();
                Collection term = result.getTerm();
                Term t;
```

---

[5] Note that for shared/composite aggregations, the initial keyword for the reference is necessary in case of more than one reference (with different roles) to the same (aggregated) class.

```
}{ "{" t=Term() { term.add(t); }
    ( "," t=Term() { term.add(t);} )* "}"
    { /*sets the derived attribute size*/
result.setSize(term.size());}
{return result;}}
```

Example 3 By the rules above, the production rule for the Asm class in Fig. 1 can be completed as follows.

EBNF: Asm ::= "asm" ("isAsyncr")? <ID_Asm> Header (Initialization)* ("default" Initialization)?

Body ("main" RuleDeclaration)?

```
Asm Asm() : { Asm result = structurePack.getAsm().createAsm();
              String name;
              boolean isAsyncr = false;
              Header headerSection;
              Initialization initialState;
              Body bodySection;
              RuleDeclaration mainrule;
}{ "asm"  [ "asyncr" {result.setAsynchr(true);}]
    name = <ID_Asm> {result.setName(name);}]
    //reads the header and sets the header reference of the ASM
    headerSection = Header(){ result.setHeaderSection(headerSection);}
    //reads the initial states and the body of the ASM
    //reads the main rule of the ASM sets the main rule reference
    [ "main" mainrule = RuleDeclaration(){ result.setMainrule(mainrule);}]
    <EOF>
{ return result;}}
```

3.7 Generalization

We distinguish between a generalization from an *abstract* class and a generalization from a *concrete* class.

Rule 16: In case classes $C_1, .., C_n$ inherit from an *abstract* class C, the production rule for the non terminal C is a choice group C::=C$_1$|..|C$_n$. Attributes and references inherited by classes C_i from the class C are represented in the same way in all production rules for the corresponding non terminals C$_i$.

Example Fig. 3 shows the complete classification of the ASM transition rules under the abstract class *Rule*. The production rule for the non terminal Rule follows.

EBNF : Rule ::= TermAsRule | BasicRule | TurboRule | DerivedRule

In JavaCC we introduce the following method, where (...|...) denotes the choice operator.

```
Rule Rule(): { Rule result;
}{ // expansion unit
    (result = TermAsRule() | ... | result = DerivedRule )
{ return result;}}
```
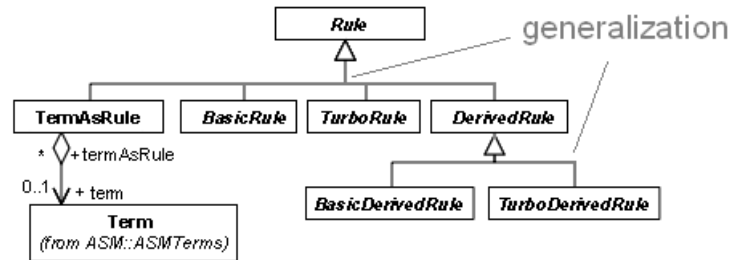
Figure 3. Example of Generalization from an abstract class

Rule 17: If classes $C_1,..,C_n$ inherit from a *concrete* class C, a production rule C::= C_c | C_1|...|C_n is introduced to capture the choice in the class hierarchy, together with a production rule for the new non terminal symbol C_c built according to the content (attributes and references) of of the superclass C. We assume that attributes and references of C inherited by classes C_i are represented in the production rules for the non terminals C_i as in that for C_c.

3.8 Constraint

OCL constraints are not mapped to EBNF concepts. Appropriate parser actions are added to the JavaCC code to instruct the generated parser on how to check whether the input model is well-formed or not according to the OCL constraints defined on the top of the metamodel. In our case, we explicitly implemented in Java an OCL checker by hard-encoding the OCL rules of AsmM. Constraint incompatibility errors are detected and reported using standard Java exception handling. Alternatively, an OCL compiler could be connected to the generated parser for the constraint consistency check.

4 The AsmM Concrete Syntax

The complete AsmM-CS grammar derived from the AsmM can be found in [7]. Fig. 4 shows the specification written in AsmM-CS of a Flip-Flop device originally presented in [9, page 47]. The first rule (FSM) models a generic finite state machine and the second rule (FLIPFLOP) instantiates the FSM for a Flip-Flop:

FSM(*i*,*cond*,*rule*,*j*) =
 if ctl_state = *i* **and** *cond* **then** {*rule,* ctl_state := *j*}   **endif**
FLIPFLOP = {FSM(0,*high*,**skip**,1),FSM(1, *low*,**skip**,0)}

5 Related Work

The OMG standardization effort for a *Human Usable Textual Notation* for the Enterprise Distributed Object Computing (EDOC) standard [16], is the main attempt to generate text grammars from specific MOF metamodels. The HUTN proposal is conceptually closer to our approach, as it aims at providing a textual input language for human

```
asm FLIP_FLOP import STDL/StandardLibrary
signature:
 domain State subsetofNatural
 controlled ctl_state : State
 monitored high : Boolean
 monitored low : Boolean
default init initial_state:
 function ctl_state = 0
 function high = false
 function low = false
definitions:
domain State = {0,1}
macro r_Fsm ($ctl_state in State, $i in State, $j in State,
             $cond in Boolean, $rule in Rule ) =
  if $ctl_state=$i and $cond then par  $rule
                                       $ctl_state := $j endpar endif
axiom over high(),low(): not( high and low )
main rule r_flip_flop = par
  r_Fsm(ctl_state,0,1,high,<<skip>>)
  r_Fsm(ctl_state,1,0,low,<<skip>>) endpar
```

Figure 4. Flip-Flop Specification

consumption (XML was explicitly excluded as being insufficiently human-friendly on a large scale). However, although automatic, the HUTN approach is designed to work with any MOF model and generate their target notation based on predefined patterns, it does not, therefore, permit a detailed customization of the generated language which reflect the object-oriented nature of the MOF meta-language.

Another MOF-to-text tool is the Anti-Yacc [10], which can be used to extract the contents of MOF-based repositories in a textual form conforming to some specified syntax. This tool is, therefore, useful to extract information from a MOFlet taking the target language grammar as input, i.e. to realize *walkers* in a MOF repository for code generation, interfacing with legacy syntaxes, and general report writing.

In [14,17], a metamodel for the ITU language SDL-2000 [24] is obtained by a semi-automatic *reverse engineering* process that allows the derivation of a metamodel from the SDL grammar definition. A very similar method to bridge *grammarware* and *mod-elware* is also proposed by other authors in [6,27]. All these approaches are complementary to our *forward engineering* process. Moreover, while we adopt a pure MOF to write metamodels, in the approaches mentioned above, especially [27], a *MOF profile* which strictly reflects the organization and grouping mechanism of the EBNF formalism is used and therefore the target metamodel obtained from a source grammar requires to be heavily processed not only (as expected) to provide the additional semantics not captured by the EBNF, but also to remove information regarding EBNF technicalities.

Defining *graphical* concrete syntaxes on the top of metamodels is another problem already addressed by the OMG with the adoption of a standard for diagram interchange for UML2 (UML-DI) [26] and by numerous authors (see for example [25]) and meta-

CASE tools like GME [13], DOME [12], AToM3 [11], MetaCASE [19], etc. Indeed, it has to be considered different from the problem addressed in this paper, although related in goal – both concern the definition of concrete syntaxes of languages whose abstract syntax is already available in form of a metamodel.

Concerning the case-study, other previous proposals exist for the definition of a textual notation for ASMs. The AsmL [20] developed by the Foundation Software Engineering group at Microsoft is the greatest effort in this respect. AsmL is a rich executable specification language, based on the theory of ASMs, expression- and object-oriented, and fully integrated into the .NET framework. However, AsmL does not provide a semantic structure targeted for the ASM method. "One can see it as a fusion of the Abstract State Machine paradigm and the .NET type system, influenced to an extent by other specification languages like VDM or Z" [30]. Adopting a terminology currently used in the MDA vision, AsmL is a platform-specific modeling language (PSM) for the .NET type system. A similar consideration can be made also for the AsmGofer language [23]. An AsmGofer specification can be thought, in fact, as a specific PSM for the Gofer environment.

6 Conclusions

this paper, we propose a MOF-to-EBNF mapping to derive a *textual* concrete syntax from a language's metamodel (the abstract syntax). We also give guidelines helping a language designer to instruct, in a semi-automatic way, a traditional parser generator for grammars (like JavaCC) to generate a *metamodel-specific bridge* between the grammarware and the modelware technical spaces. We believe that the MOF is suitable as meta-language for deriving different concrete notations from metamodels, however, further investigations are necessary to decide if arbitrary grammars can be generated.

We implemented the approach for the definition of an EBNF grammar from the ASM metamodel for a textual notation for the ASM formal method. We also provide a parser which processes specifications written in the concrete syntax, checks for their consistency with the metamodel, and translates information about concrete models into a MOF-based instance repository. The applicability of our results to make possible the coupling of different types of ASM tools has been discussed in [22,15]. Soon, we plan to integrate the compiler with a proper Integrated Development Environment which acts as front-end for the modeler, and to develop an ASM virtual machine to simulate AsmM-CS specifications. Furthermore, we intend to upgrade the AsmM to MOF 2.0 and to use the Mof2Text standard or the QVT standard to specify the rules in a more formal way. We are also evaluating the possibility to exploit other metamodelling frameworks, like the ATL project [2], the MTL engine [4], the Xactium XMF Mosaic [5], to better support *model transformations* and *model evolution activities* such as code generation, reverse engineering, model refinement, model refactoring, etc..

References

1. Java Compiler Compiler - The Java Parser Generator. `https://javacc.dev.java.net/`.

2. The ATL model transformation language. `http://www.sciences.univ-nantes.fr/lina/atl/`.

3. The MDR (Model Driven Repository) for NetBeans. `http://mdr.netbeans.org/`.

4. The MTL Engine. `http://modelware.inria.fr/`.

5. The Xactium XMF Mosaic tool suite. `http://www.modelbased.net/www.xactium.com/`.

6. M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical report, Turku Centre for Computer Science, 2003.

7. The Abstract State Machines Metamodel (AsmM) website. `http://www.dmi.unict.it/~scandurra/AsmM/`.

8. J. Bézivin. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2):21–24, 2004.

9. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

10. D. Hearnden and K. Raymond and J. Steel. Anti-Yacc: MOF-to-text. In *EDOC*, pages 200–211, 2002.

11. J. de Lara and H. Vangheluwe. Using AToM^3 as a Meta-Case Tool. In *ICEIS*, pages 642–649, 2002.

12. Honeywell. Dome users guide. 2000.

13. M. J. Emerson, J. Sztipanovits, and T. Bapty. A MOF-Based Metamodeling Environment. *j-jucs*, 10(10):1357–1382, 2004.

14. J. Fischer, M. Piefel, and M. Scheidgen. Using Metamodels for the definition of Languages. In *Proc. of Fourth SDL And MSC Workshop (SAM04)*, 2004.

15. A. Gargantini, E. Riccobene, and P. Scandurra. AsmM the Abstract State Machines matamodel. Technical report, University of Bergamo - Department of Management and Information Technology, 2006.

16. OMG, Human-Usable Textual Notation, v1.0. Document formal/04-08-01. `http://www.uml.org/`.

17. J. Fisher and E. Holz and A. Prinz and M. Scheidgen. Toolbased Language Development. In *Proc. of Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04)*, 2004.

18. Java Metadata Interface Specification, Version 1.0, 2002.

19. MetaCASE. ABC to MetaCASE Technology. White paper, 2004.

20. M. R. F. of Software Engineering Group. AsmL: The Abstract State Machine Language. `http://research.microsoft.com/foundations/AsmL/`.

21. Poseidon UML Tool. `http://www.gentleware.com`.

22. E. Riccobene and P. Scandurra. Towards an Interchange Language for ASMs. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines. Advances in Theory and Practice*, LNCS 3052, pages 111 – 126. Springer, 2004.

23. J. Schmid. AsmGofer. `http://www.tydo.de/AsmGofer`.

24. SDL (Specification and Description Language. ITU Reccomandation Z.100. `http://www.itu.int`.

25. Thomas Baar. Making Metamodels Aware of Concrete Syntax. In *Proc. of ECMDA'05, LNCS Vol. 3748*, pages 190–204, 2005.

26. OMG. UML 2.0 diagram interchange specification, ptc/03-09-01.

27. M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005)*, Montego Bay, Jamaica, 2005.

28. OMG, XML Metadata Interchange (XMI) Specification, v1.2.

29. W3C, The Extensible Markup Language (XML).

30. Y. Gurevich and B. Rossman and W. Schulte. Semantic Essence of AsmL. Microsoft Research Technical Report MSR-TR-2004-27, March 2004 .

PIM to PSM transformations for an event driven architecture in an educational tool

Geert Monsieur[1], Monique Snoeck[1], Raf Haesen[1,2], Wilfried Lemahieu[1]

[1]KULeuven, Faculty of Economic and Applied Economic Sciences
Management Information Systems Group
Naamestraat 69; 3000 Leuven, Belgium
[2] Campus Vlekho
Koningsstraat 336, 1030 Brussel, Belgium
{Geert.Monsieur, Monique.Snoeck, Raf.Haesen, Wilfried.Lemahieu}@econ.kuleuven.be

Abstract. This paper presents experiences with the use of an MDA approach to generate prototype applications from a conceptual domain model in the context of teaching object-oriented domain modelling. Each conceptual model used to generate the prototype consists of a combination of three views (a class diagram, a proprietary object-event table and a group of finite state machines) and constitutes as such the platform-independent model (PIM). We describe in detail how our event-driven PIM is transformed into an event-driven platform-specific model which is almost directly used to generate the running prototype application (the code). We conclude with a discussion of the lessons we learned, problems we faced, potential solutions and critical aspects for a successful MDA story.

1 Introduction

This paper presents experiences with applying the MDA-approach in an educational context, namely in the context of teaching object-oriented business domain modelling[1]. The main goal of the course is to familiarise students with object-oriented analysis techniques and to let them acquire the necessary skills to actually perform domain modelling. In addition, students should be able to envisage the mutual relationship and influence between the IT-system and the organisational work system. For example, the student should learn to evaluate the implications of differences between models both from an IT and an organisational perspective.

The student's prior capability is a crucial element to achieve the educational goal. The course is attended by a set of students with a large variation in preliminary knowledge of object-oriented concepts and programming skills. One can roughly identify three major groups of students. One group of students are very much technology oriented and have good object-oriented programming skills. Their IT-orientation is sometimes a disadvantage as it hampers them in taking a more business oriented view towards specifications: they tend to think in a solution-oriented mode, while the goal of the

---

[1] The course's page can be found on http://mermaid.econ.kuleuven.be/content.aspx/

course is to focus on the characteristics of the problem world. The second group of students have a substantial set of business-oriented skills, are skilled in data modelling and have basic skills in object-oriented programming. This is the group of students that is most at ease with the presented material. However, because of their preliminary database course, these students tend to think in a purely data-oriented way, and have difficulties in conceptualizing the behavioural aspects of an object-oriented model as defined in class operations and finite state machines (FSM). Finally, for a third group of students, this course is their first contact with object-orientation. They have no programming skills and have not yet followed a course on database modelling. Especially this group has difficulties of imagining the concrete system that will result out of an abstract model.

One of the major goals is that the student should be able to form a mental model of a concrete information system built from the conceptual model he/she made. We can describe a mental model as a model that gives a clear concretisation of the working and behavioural aspects of an information system. One of the techniques to concretise models is prototyping, a widely-spread approach for validating user requirements. However, whereas any student with some crafting skills can build a prototype of a house with simple tools such as a cutter, card board and glue, building prototypes of information systems requires a reasonable amount of programming skills and time. Given the fact that the majority of students have no or but limited programming skills, requiring students to build a prototype for every single model they make is impossible. Nevertheless, models only are too abstract and many students find it difficult to understand the consequences implied by a difference between two potential models for a same set of requirements.

Many tools are able to generate fairly easily simple data-oriented prototypes providing standard create, update and delete features per business object type (e.g. using wizard-generated forms in MS Access). However, the goal of the course goes beyond data modelling: students should be able to conceptualise the behavioural aspects and interaction aspects of business objects as well. For generating behavioural aspects, UML advocates the use of finite state machines to model behavioural aspects of a class and collaboration diagrams or interaction diagrams to model the interaction aspects. While the latter diagrams can still be considered as platform independent models, they are at a quite low level of abstraction, since they model software behaviour at the level of the individual message exchanges. Domain modelling is inherently much more abstract. Whereas the life-cycle of a business object can be modelled at a sufficiently high level of abstraction by means of a finite state machine, the design of message exchanges between objects is more oriented towards the solution space than to the problem space. As an example, the life cycle of an order in terms of its creation, confirmation, shipping and payment is at a sufficiently high level of abstraction. The fact that the creation of an order requires the verification of the customer's status, the availability of the product the planning of the shipping will require some interaction between several domain classes. The exact sequence of the processing is however beyond the scope of domain modelling. It is the task of the *designer* to decide the best interaction schema to realise the creation of an order. As a result, one of the goals of our research is to devise a standard pattern for modelling and implementing domain object interaction.

An additional goal is to develop a modelling and implementation approach that is scalable to real life systems.

A final consideration is the user interface of the generated prototype. Similarly as for domain interaction, the modelling of user-system interaction is beyond the scope of domain modelling. Hence, also for the user interface, we need to resort to a pattern-based approach. In this case however, scalability to real-life system was not part of our goal.

2 Solution

Students create a platform independent model of the business domain. The tool will then map this PIM to a PSM and code, using a set of standard patterns. In this section, we first describe the concepts of our PIM, subsequently followed by a brief discussion of our PSM and we conclude this section with an overview of the rules we defined for the PIM-to-PSM transformation.

2.1 The platform independent model (PIM)

The platform independent model consists of a class diagram, an interaction model and a number of finite state machines. We decided to employ the widely adopted Unified Modelling Language (UML) as the modelling notation for all diagrams. This way, it is easy to interoperate with other tools using the XML Metadata Interchange (XMI) format. Additionally, most MDA frameworks support the use of UML, such as the AndroMDA framework[2] we used to implement our solution.

The class diagram is a restricted form of UML class diagram: the types of associations are limited to binary associations, with a cardinality of 1-to-many or 1-to-1. Many-to-many associations need to be converted to an intermediate class. In theory, this conversion could be realised as a transformation as well. For didactic and methodological purposes, students are required to transform these associations manually: it obliges them to think about the constraints that apply to this association class (see [12] for a detailed motivation of this approach).

Figure 1 shows a class diagram for a sales domain that will be used throughout the rest of this paper as running example.
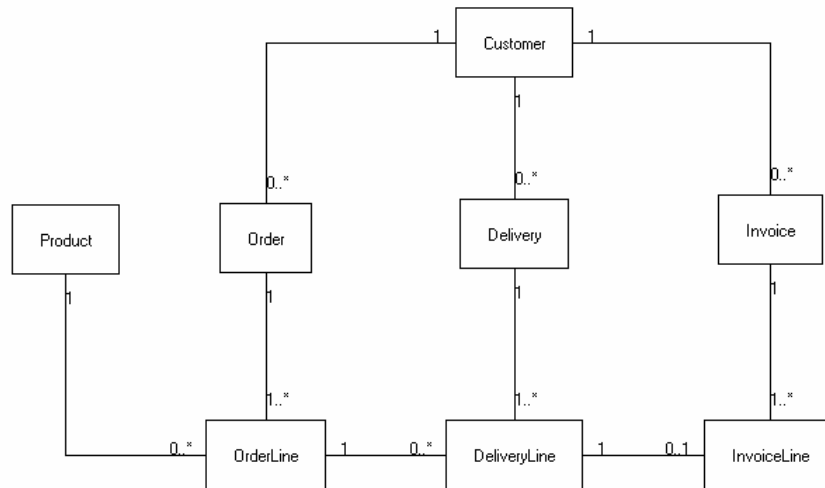
---

[2] http://www.andromda.org

Fig. 1. A class diagram for the sales example

Each class has a number of operations that are classified as creators, modifiers or ending operations. In a typical object-oriented model, classes interact by invoking each other's operations. As explained in the previous section, domain modelling is at a high level of abstraction and oriented to the problem space rather that to the solution space. When an action requires processing in several domain classes, we do not want the modeller to design the details of the collaboration between the classes, as this pertains to the solution space rather than to the problem space. We therefore developed a specific modelling technique which allows identifying atomic actions and indicating which classes are involved in the processing of that action. The atomic actions are called 'business events' to stress their real-world (=business) and atomic (=event) character. The concept of 'business event' is very similar to the concept of event as used in Syntropy [2] and action in Catalysis [3]. In the Object-Event Table (OET) the requirements engineer indicates which domain object type is involved in the processing of a given business event. This processing can be the validation of the action, the creation, the modification or the ending of an instance of that domain object type, respectively indicated with V, C, M or E in the table (see table 1). As an example, creating an order may require the domain object type Customer to verify the customer status (blacklisted or not), and the object type Order to create an instance of that class. Adding an order line to that order to order a product, will require the domain object type Product to verify the availability of the product and update the stock-level of that product, will require the domain object type OrderLine to create an instance of that class and will require the domain object type Order to add the created order line to the order instance.

	Customer	Product	Order	OrderLine	...
cr_order	V		C		
cr_orderLine		V+M	M	C	
....					

Table 1. Partial Object-Event table for the Sales example

Obviously the OET is not a default modelling technique supported by the UML. The implementation of a business event requires the elaboration of a collaboration schema that specifies the required interactions to process the action.   If a standard collaboration pattern can be defined, this can be used to automate the transformation of the OET to equivalent UML concepts that are used to generate a platform specific model.

Finally, the platform independent model of the domain contains one or more finite state machines per domain object type. The finite state machines allow the object type to impose sequence constraints on the business events it is involved in. Multiple Finites State Machines allow to model independent aspects as parallel machines. Before transformation, these are algorithmically transformed to a single Finite State Machine that represents the parallel composition of all specified finite state machines for that object type. Figure 2 shows an example of a FSM for the object type Product.
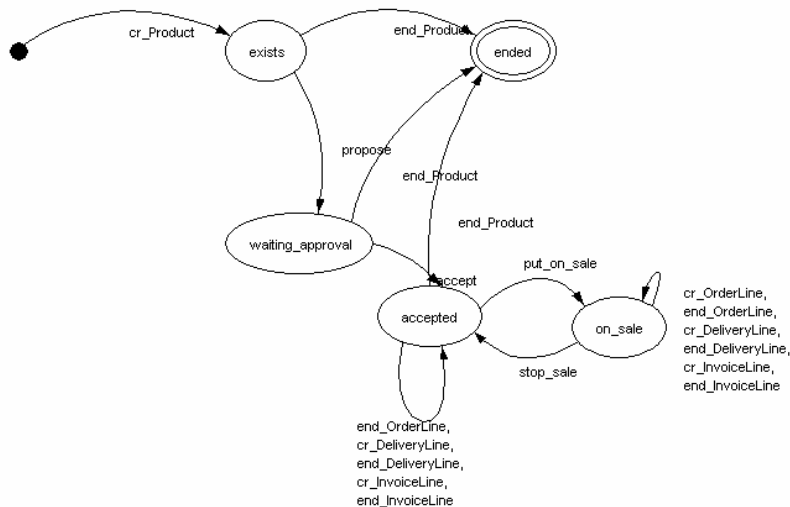
Fig. 2. Finite State Machine for Product
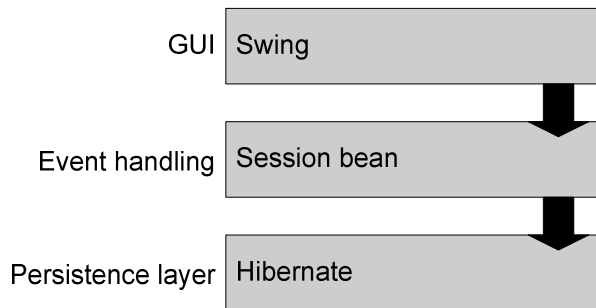
2.2 The platform specific model (PSM)

Fig. 3. The three layers of the platform specific model

The target architecture consists of three layers, a graphical user interface (GUI) layer, an event handling layer and a persistence layer (see figure 3). Although any object-oriented programming languages would provide us with the required programming mechanisms, we chose the J2EE framework to implement the solution.

The graphical user interface (based on the Java Swing architecture) has only basic functionality like triggering the creating and ending of objects, and triggering other business events. The GUI layer is built on top of the event handling layer. The task of the latter layer is to handle all events correctly by managing the appropriate interactions with the objects in the persistence layer.

2.3 Transformation rules

We will now discuss our target architecture in detail by focusing on the PIM-to-PSM transformations. In the graphical representations of the transformation rules we use white boxes for PIM concepts, dark gray boxes for the PSM concepts and light gray arrows for the transformation.

Persistence layer
This layer forms the basis of our target architecture. All other layers in our architecture will make use of the services provided by the persistence layer. Generation of this layer is made possible by transforming three PIM elements, namely the class diagram, the columns of the object event table and the finite state machines.

Transforming the class diagram and operations (columns of OET)
For transforming the class diagram and operations we have defined the following transformation rules:

Each object type in the PIM is transformed into (see figure 4):
- an *abstract class* and *implementation class* in our PSM. By making a distinction between an abstract and implementation class it is easier to see the difference between generated PSM elements and PSM elements added

manually after the PIM-to-PSM transformation took place. This leads to a better traceability which can be important for managing the complexity of MDA [6]. Furthermore we use the object-relational mapping (ORM) technique of Hibernate[3] to make this type of objects persistent.

- A *factory* which makes it easier to create and collect objects of the same type [4].
- One *method* (in the mapped abstract class) for each event which the object type participates in and one *method* for checking the corresponding preconditions. The (columns of the) Object-event table contain(s) the information needed for this transformation rule.
  Pseudo code:
  `method_for_event()` and `check_preconditions_for_event()`
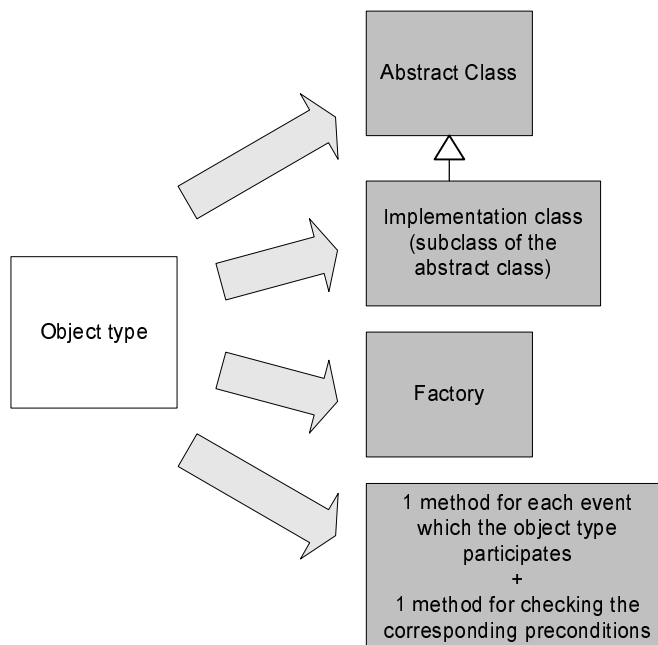
Fig. 4. Transformation rules for an object type

Each attribute of an object type is transformed into an attribute of the mapped class (see figure 5).
Each one-to-many association in our class diagram is transformed into an association attribute in the mapped classes (see figure 6).

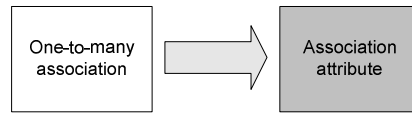[3] http://www.hibernate.org

```
┌──────────────┐      ┌──────────────┐   ┌──────────────┐      ┌──────────────┐
│ Attribute of │ ==>  │ Attribute of │   │ One-to-many  │ ==>  │ Association  │
│ Object type  │      │ mapped class │   │ association  │      │  attribute   │
└──────────────┘      └──────────────┘   └──────────────┘      └──────────────┘
```
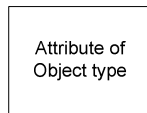
Fig. 5.   **Fig. 6.**

Transforming the finite state machines
To transform the finite state machines we created the following transformation rules:

An object type's finite state machine is transformed into (see figure 7):

- *An abstract state class* (associated with the mapped class) with *state subclasses* for each state in the FSM: this is based on the state pattern [4].
- *Methods in the state subclasses* for checking state conditions for each event which the object type participates in.

```
                          ┌────────────────────┐
                          │ Abstract state class│
                    ==>   │ (associated with    │
                          │  mapped class)      │
                          └────────────────────┘
┌──────────────┐          ┌────────────────────┐
│ Object type's│   ==>    │ For each state of the│
│     FSM      │          │  FSM a State        │
│              │          │  subclass of mapped │
└──────────────┘          │  abstract state class│
                          └────────────────────┘
                          ┌────────────────────┐
                    ==>   │ Per mapped state    │
                          │ subclass methods    │
                          │ for checking state  │
                          │ conditions for each │
                          │ event which the     │
                          │ object type         │
                          │ partcipates in      │
                          └────────────────────┘
```
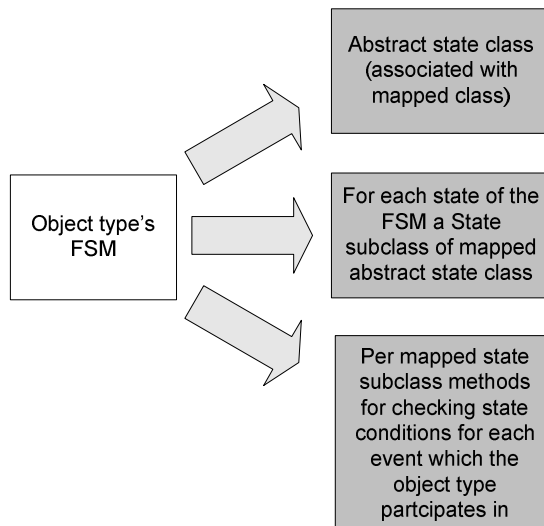
Fig. 7. Transforming the Finite State Machines

Event handling layer
The event layer can be seen as the heart of our *event-driven* target architecture. Good organised mechanisms to handle events play a central role in an event-based PSM. For this purpose we currently use one session bean which bundles all needed event handlers, although it is possible to use another way of transforming the event-driven concepts, e.g. by using one PSM class per event. This would of course lead to a different transformation and a different PSM.

Transforming the events (rows of OET)
The event layer described in this paper consists of only one session bean which *handles* the events. In general the following transformation rule is valid for our PSM: For each event (found as rows of the OET) there is an operation (an event handler) in a so called EventHandlerBean (see figure 8). The implementations of these event handlers are also generated. The latter generation follows a *standard collaboration pattern*. We will now discuss further details about this pattern.
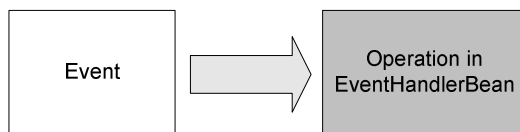
Fig. 8. Transforming the events

Standard collaboration pattern for generation of handle_event()

We summarize the pattern in four steps. Steps 1 and 2 are represented in figure 9. Steps 3 and 4 are visualized in figure 10.
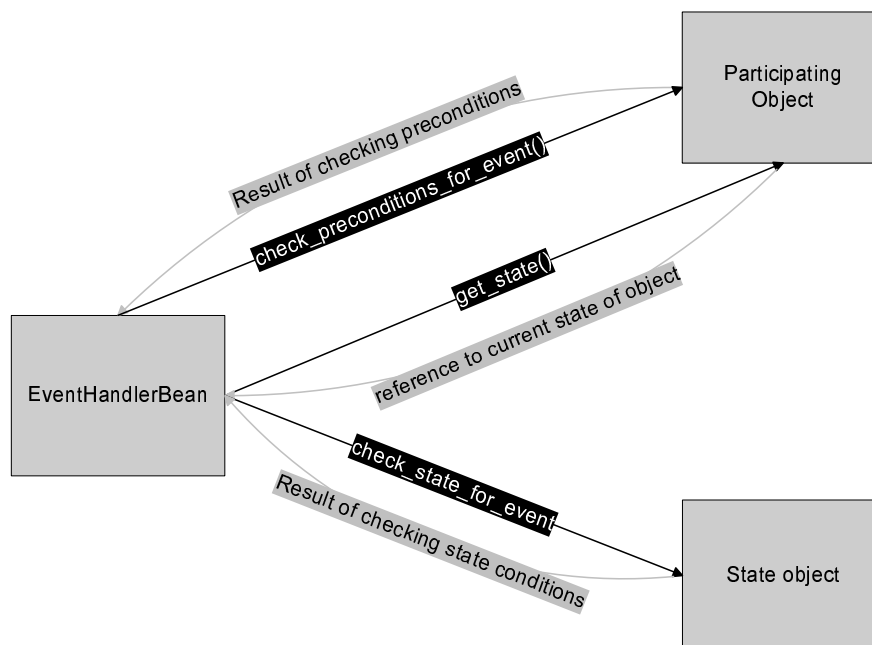
Fig. 9. Standard collaboration pattern for generation of handle_event() (step 1 and 2)

Step 1 The event handler 'asks' every participating object (an object which is involved in the processing of a business event) whether all preconditions set by the object are met. These preconditions can be defined by the analyst (as V's in the OET), but also include conditions that can be derived from other parts of the model. For example, associations between classes will lead to preconditions to maintain referential integrity.

Pseudo code:
```
participant_1.check_preconditions_for_event()
participant_2.check_preconditions_for_event()
…
participant_n.check_preconditions_for_event()
```

Step 2 Similarly to the previous step the event handler retrieves from every participating object its current state (or reference to the corresponding state object) and checks whether that state allows further processing of the event.

Pseudo code:
```
participant_1.getState().check_state_for_event()
participant_2.getState().check_state_for_event()
…
participant_n.getState().check_state_for_event()
```

Step 3 If all results of the tasks in step 1 and 2 are positive (this means 'no exceptions are thrown'), the event handler invokes the methods in the participating objects which correspond with the triggered event (i.e. the C's, M's and E's in the OET).

Pseudo code:
```
participant_1.method_for_event()
participant_2.method_for_event()
…
participant_n.method_for_event()
```

Step 4 Next, (if all results of the tasks in step 1 and 2 are positive) the event handler executes in all state objects retrieved in step 2 the method for modifying the state (according to the triggered event).

Pseudo code:
```
participant_1.getState().change_state_for_event()
participant_2.getState().change_state_for_event()
…
participant_n.getState().change_state_for_event()
```
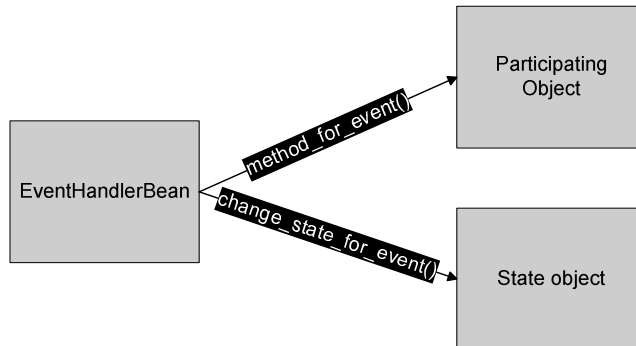
Fig. 10. Standard collaboration pattern for generation of handle_event() (step 3 and 4)

In case of negative results in step 1 and/or 2 an exception is thrown. In the prototype application the end-user will notice this exception as a message box which tells what went wrong (violated preconditions or state conditions) and why the processing of the event is not allowed.

GUI layer
The design of the GUI layer is based on a pattern for event-based user interface proposed in [10]. Although the GUI layer is like other parts of our PSM strongly based on the event-based aspects of our PIM (e.g.: one window generated per defined event, etc.), we consider the discussion of these transformation rules as beyond the scope of this paper.

2.4 Transformation rules applied for the sales example

In this section we show (some part of) the results from applying the transformation rules to the sales example described earlier in this paper in the discussion of our PIM concepts.

Generated persistence layer
In figure 11 the generated persistence layer is partially shown (to maintain the overview only two transformed object types are shown, and some methods are hidden).
The transformed class diagram and operations (columns of the OET) are visualized as white boxes. One can see that object type OrderLine is transformed into an abstract class OrderLine, an implementation class OrderLineImpl and a factory OrderLineFactory. The attributes and methods to check preconditions and process events are also generated for the OrderLine class. The required methods for the Product class are not shown to avoid overloading the figure with too much information.
The gray boxes in figure 11 represent the transformed finite state machines. The different states for Product are transformed in different subclasses (ProductExistsState, ProductWaiting_ApprovalState, …) of an abstract state class

(ProductState). Only a few generated methods for checking state conditions and changing states are shown in this figure.
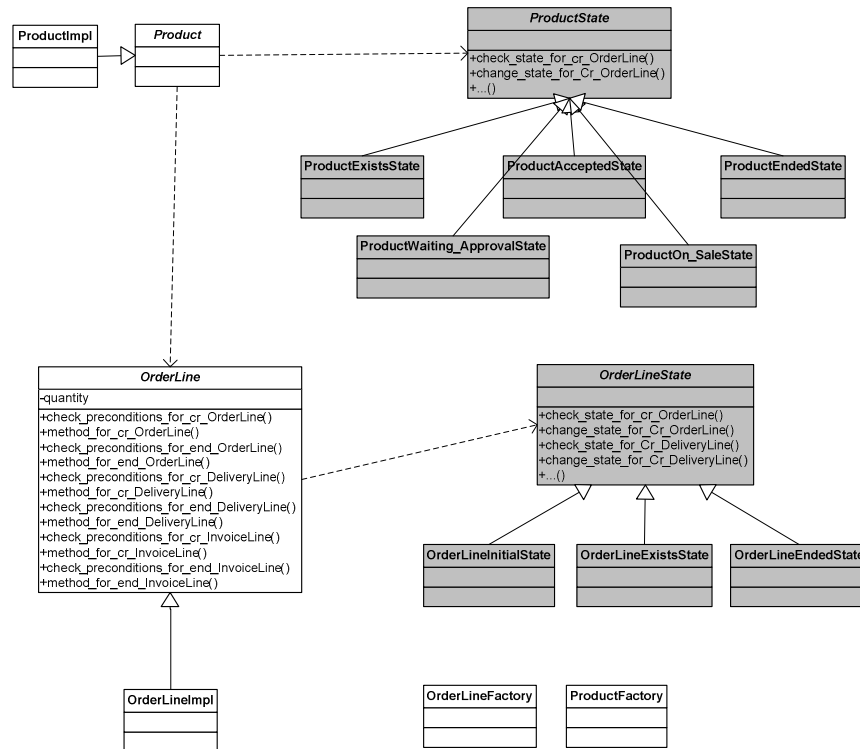

Fig. 11. Partial generated persistence layer

Generated event layer

At first sight the generated event layer looks very straight forward (see figure 12). However the real challenge is the implementation of the different event handlers defined in the session bean. We give an example in which we apply the collaboration pattern for the event cr_OrderLine.

```
┌─────────────────────────────┐
│ EventHandlerSessionBean     │
├─────────────────────────────┤
│                             │
├─────────────────────────────┤
│ +handle_cr_Customer()       │
│ +handle_end_Customer()      │
│ +handle_cr_Product()        │
│ +handle_end_Product()       │
│ +handle_propose()           │
│ +handle_accept()            │
│ +...()                      │
│ +handle_cr_OrderLine()      │
│ +...()                      │
└─────────────────────────────┘
```

Fig. 12. Generated event layer

Standard collaboration diagram applied for event cr_OrderLine()

Involved objects:
```
orderLine, order, product
```
Pseudo code for handle_cr_OrderLine()[4]
```
// checking preconditions (step 1)
orderLine.check_preconditions_for_cr_OrderLine()
order.check_preconditions_for_cr_OrderLine()
product.check_preconditions_for_cr_OrderLine()
// checking state conditions (step 2)
orderLine.getState().check_state_for_cr_OrderLine()
order.getState().check_state_for_cr_OrderLine()
product.getState().check_state_for_cr_OrderLine()
// event processing (step 3)
orderLine.method_for_cr_OrderLine()
order.method_for_cr_OrderLine()
product.method_for_cr_OrderLine()
// state modifications (step 4)
orderLine.getState().change_state_for_cr_OrderLine()
order.getState().change_state_for_cr_OrderLine()
product.getState().change_state_for_cr_OrderLine()
```

3 Conclusions

3.1 Summary

This article has presented an approach to derive prototype applications from conceptual domain models using the MDA ideas. Each conceptual model consists of a

---

[4] Actually a call to OrderLineFactory is required before the execution of step 1 is possible.

combination of three views and constitutes as such the platform-independent model. The first view presents the business object types and their interrelations by means of a class diagram. The dynamic view consists of a proprietary object-event table that identifies business event types and indicates how a business event type affects object types. If an object type is affected by an event type it will define a method implementing the effect of this event type on objects of this type. Finally a set of finite state machines constrain the invocation of the business events.

Using this platform-independent model it is possible to generate three-tier prototype applications. The persistence layer consists of the business object types enriched with all required method types derived from the OET. On top of that an event handling layer is constructed using the OET and the finite state machines. For each event type the user-defined preconditions and state preconditions of the participating object types are checked. If all preconditions are met, the appropriate methods and state transitions are executed. The GUI layer provides access to the events that can be triggered.

3.2 Lessons learned: what worked fine, remaining problems, potential solutions and future work

After introducing our tool in an academic course, we noticed that for students, it is not always clear what's wrong: their model or the code-generator. This is an important lesson: the more we are going to work according to the principles of the model-driven architecture, the more important it will be to ensure the quality of our implementation, and in casu of the model transformations. If the final application doesn't work as expected (either because of bugs or because it doesn't behave as expected) there are two possible sources for the error: either the model (PIM) is wrong, or the transformation rules are not like they should be. To support the process of making a *high-quality* PIM - which can be relatively easily transformed into a PSM and a working prototype application - we enforce the students making *consistent* models. This enforcement is realized by means of advanced (intra-model and horizontal) consistency management techniques like consistency by construction, monitoring, analysis, etc. [5][14].

We also have experienced that managing our transformation rules becomes a crucial issue for a successful MDA story. This should be improved in our future implementations by making better use of standards like MOF and QVT [9].

At the moment we can't generate all information modelled in our PIM. For example mandatory relationships are not enforced in the generated application, and transforming inheritance relationships is still not possible. Transformation rules for inheritance still need to be defined. The use of inheritance in the OET in combination with certain constraints can yield complex models that are very difficult to translate to PSMs in an automated way. Discovering transformation rules that are in general robust to arbitrary combinations of PIM concepts is quite a challenge for the future.

The concept of business events plays an important role in our PIM and should make it possible to define transformation rules to generate other PSM than the one described in this paper. We intend to generate transformations for service-oriented and component-based platforms, because it's already proven that events can also add

value (events as contract, coordination 'tool', etc.) in these sorts of platforms [7][13]. Notice that although the event handler shows a striking similarity with a number of event based architectures [1][8][11], there are substantial differences. For example, the fact that an event is verified prior to the notification to participating domain objects makes it very different from a classical fire-and-forget architecture. For a detailed comparison, the reader is referred to [13].

References

[1]   Barrett, Daniel J., Clarke Lori A., Tarr, Peri L., Wise, Alexander E. (1996), *A Framework for Event-Based Software Integration*, ACM Transactions on Software Engineering and Methodology, 5(4), 1996, pp. 378-421

[2]   Cook S. and Daniels J. (1994) *Designing object systems: object-oriented modeling with Syntropy*, Prentice Hall

[3]   D'Souza D. and Wills A.C (1999): *Objects, Components, and Frameworks with UML, the Catalysis approach*, Addison Wesley, Reading MA

[4]   Gamma E., Helm R., Johnson R. and Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5]   Haesen R. and Snoeck M.: *Implementing Consistency Management Techniques for Conceptual Modeling*, accepted for UML2004: 7th conference in the UML series, Lisbon, Portugal, October 10-15, 2004

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

[7]   Lemahieu W., Snoeck M., Goethals F., De Backer M., Haesen R., Vandenbulcke J. and Dedene G.: *Coordinating COTS Applications via a Business Event Layer*. IEEE Software, 22(4):28-35, 2005

[8]   Meier R., Cahill V. (2002) *Taxonomy of Distributed Event-Based Programming Systems*. Technical Report, TCD-CS-2002, Dept. of Computer Science, Trinity College Dublin, Ireland

[9]   MOF and QVT, OMG, http://www.omg.org/mda/specs.htm

[10]   Robinson K. and Berrisford G.: *Object-oriented SSADM*. Wiley Chichester, 1994

[11]   Shaw M., Garlan D. (1996), *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, N.J., 242 pp.

[12]   Snoeck M. and Dedene G.: *Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types*. IEEE Trans. Software Eng., 24(4):233-251, 1998.

[13]   Snoeck M., Lemahieu W., Goethals F., Dedene G. and Vandenbulcke J. (2004), *Events as Atomic Contracts for Application Integration.* Data and Knowledge Engineering 51, 81-107

[14]   Snoeck M., Michiels C. and Dedene G.: *Consistency by construction: the case of MERODE*, in Jeusfeld, M. A., Pastor, O., (Eds.) Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003, Proceedings, 2003 XVI, 410 p., Lecture Notes in Computer Science, Volume 2814, pp.105-117

Query/View/Transformation Language for Multidimensional Modeling of Data Warehouses

Jose-Norberto Mazón and Juan Trujillo

Dept. of Software and Computing Systems
University of Alicante, Spain
{jnmazon,jtrujillo}@dlsi.ua.es

Abstract. Nowadays, it is widely accepted that the development of data warehouses (DW) is based on multidimensional (MD) modeling. In the past few years, several approaches have been proposed for representing the main MD properties in a conceptual model. These approaches also define a set of informal guidelines to derive a logical representation tailored to a specific technology (normally a relational implementation based on the star schema and its variants). However, these approaches do not define formal transformations in order to univocally and automatically derive every possible logical representation from the conceptual model. To overcome this problem, in this paper, we present a Model Driven Architecture (MDA) approach for the MD modeling of DWs. First, we describe how to build Platform Independent Models (PIM) and Platform Specific Models (PSM) for MD modeling by using an extension of the Unified Modeling Language (UML) and the Common Warehouse Metamodel (CWM) respectively. Then, transformations between these models are formally established by using the Query/View/Transformation (QVT) language.

1   Introduction

Data warehouse (DW) systems provide companies with many years of historical information for the success of the decision-making process. Both practitioners and researchers agree that the development of these systems must be based on multidimensional (MD) modeling [1, 2] which structures information into facts and dimensions. A fact contains interesting measures (fact attributes) of a business process (sales, deliveries, etc.), whereas a dimension represents the context for analyzing a fact (product, customer, time, etc.) by means of dimension attributes hierarchically organized. A set of fact measures is based on a set of dimensions that determine the granularity adopted for representing facts. Due to space constraints, we refer reader to [3] for a further explanation of the different aspects of MD modeling.

Several authors [3–5] have proposed various approaches for the conceptual design of DWs. These proposals try to represent the main MD properties at the conceptual level with special emphasis on data structures (i.e. facts and dimensions) by abstracting away details of the target platform where the DW will be

implemented (i.e. DataBase Management System or DBMS). Two technologies are considered in a DBMS in order to support DW design [6]:

– Relational approaches store MD data by using relational database technology. Normally, the logical representation of the MD model is a star schema (or some of its variants) consisting on a set of dimension tables and a central fact table [1]. The advantage of a relational system is scalability, since a huge amount of data can be stored.
– Multidimensional approaches store data on MD files or arrays in a more natural way. The logical representation of the MD model is represented by using cells (i.e. facts) of MD arrays (i.e. dimensions). Multidimensional systems generally provide more space-efficient storage as well as faster query response times.

Traditionally, conceptual approaches for DWs also define a set of informal guidelines to derive a logical representation tailored to one of the above-described technologies (normally a relational representation based on the star schema and its variants). However, these approaches do not define formal transformations in order to univocally and automatically derive every possible logical representation from the conceptual model.

In order to overcome this problem, in this paper, we present a Model Driven Architecture (MDA) approach for the MD modeling of DWs. First, we present how to develop a Platform Independent Model (PIM) for MD models by using our UML (Unified Modeling Language) profile [7]. This PIM is considered the conceptual model of the DW. Then, different Platform Specific Models (PSMs) can be developed as logical models depending on the underlying database technology. Different packages of the Common Warehouse Metamodel (CWM) are used to define each PSM. Later, we present a set of QVT transformation rules in order to formally obtain every PSM from PIM.

In a previous work [8], we have developed an MDA framework for the development of DWs, and we have developed transformations to obtain a PSM according to a relational database approach. In this paper, we focus on deriving a PSM related to a multidimensional database technology.

The remainder of this paper is structured as follows. Section 2 presents an overview of related work. A brief overview of MDA and QVT is given in section 3. Section 4 describes our MDA approach for MD modeling of DWs. An example is provided in section 5 to better show how to apply MDA and its transformations to the MD modeling of the DWs. Finally, in section 6 we point out our conclusions and sketch some future works.

2 Related work

Various approaches for the conceptual design of DWs[1] have been proposed in the last few years [3–5]. However, these approaches lack in formal transformations

---

[1] We refer reader to [9] for a detailed comparison.

to automatically obtain the logical representation of the conceptual DW model, and they only give some informal guidelines to undertake this task.

On the other hand, MDA has been successfully applied to several application domains, such as web services [10] and web applications [11], development of user interfaces [12], real-time systems [13], multi-agents systems [14], and so on. However, to the best of our knowledge, only one effort has been developed for aligning the design of DWs with the general MDA framework, the Model Driven Data Warehousing (MDDW) [15]. This approach is based on CWM [16], which is a metamodel definition for interchanging DW specifications between different platforms and tools. Basically, CWM provides a set of metamodels that are comprehensive enough to model an entire DW including data sources, ETL processes, MD modeling, relational implementation of a DW, and so on. These metamodels are intended to be generic, external representations of shared metadata. The proposed MDDW is based on modeling a complete DW by using elements from various CWM packages. However according to [17], CWM metamodels are (i) too generic to represent all peculiarities of MD modeling in a conceptual model (i.e. PIM) and (ii) too complex to be handled by both final users and designers. Therefore, we deeply believe that it is more reliable to design a PIM by using an enriched conceptual modeling approach easy to be handled (e.g. [7]), and then transform this PIM into a CWM-compliant PSM in order to assure the interchange of DW metadata between different platforms and tools.

3   MDA and QVT

Model Driven Architecture (MDA) is an Object Management Group (OMG) standard [18] that addresses the complete life cycle of designing, deploying, integrating, and managing applications by using models in software development. MDA separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. Thus, MDA encourages specifying a Platform Independent Model (PIM) which contains no information specific to the platform or the technology that is used to realize it. Then, this PIM can be transformed into a Platform Specific Model (PSM) in order to include information about the specific technology that is used in the realization of it on a specific platform. In fact, several PSMs can be derived from one PIM. Each of these PSMs are tailored to different platforms. Later, each PSM is transformed into code to be executed on each platform.

PIMs and PSMs can be specified by using any modeling language, but typically MOF-compliant languages as UML or CWM are used since they are standard modeling languages for general purpose and, at the same time, they can be extended to define specialized languages for certain domains (i.e. metamodel extensibility or profiles).

Nowadays the most crucial issue in MDA is the transformation between a PIM and a PSM [18]. Thus, OMG defines MOF 2.0 Query/View/Transformation (QVT), an approach for expressing these MDA transformations [19]. This is a

standard for defining transformation rules between MOF-compliant models (e.g. UML models). This standard defines a hybrid specification for transformations. On the one hand, there is a declarative part, which provides mechanisms to define transformations as a set of relations that must hold between the model elements of a set of candidate models (i.e. source and target models). On the other hand, QVT also describes an imperative part which defines operational mappings that extend the declarative part with imperative implementations. This part is used when it is difficult to provide a purely declarative specification of a relation.

In this paper, we focus on the declarative part of QVT. Specifically, we use the relations language which supports the specification of relationships that must hold between MOF models. A set of these relations defines a transformation between models. A relation is defined by the following elements:

- **Two or more domains**: each domain is a distinguished set of elements of a candidate model (PIM or PSM). This set of elements must be matched in that model by means of patterns. A domain pattern can be considered as a template for elements, their properties and their associations that must be located, modified, or created in a candidate model in order to satisfy the relation. The kind of relation between domains must be specified, since it can be marked as checkonly (labeled as C) or as enforced (labeled as E). When a relation is executed in the direction of a checkonly domain, then it is only checked if there exists a valid match in the model that satisfies the relationship (without modifying any model if the domains do not match); whereas for a domain that is enforced, when the domains do not match, model elements are created, deleted or modified in the target model in order to satisfy the relationship. Moreover, for each domain the name of its underlying metamodel is specified.
- **When clause**: it specifies the conditions that must be satisfied to carry out the transformation (i.e. precondition).
- **Where clause**: it specifies the condition that must be satisfied by all model elements participating in the relationship (i.e. postcondition).

Defining transformations by using the QVT language has several advantages: (i) it is a standard language, (ii) transformations are formally established and automatically performed, and (iii) transformations can be easily integrated in an MDA approach.

4 MDA for Multidimensional Modeling

In a previous work [8] we have defined an MDA framework for the development of DWs. In this framework, every part of the DW (data sources, ETL processes, etc.) is pretended to be designed by using an MDA approach in order to increase productivity, portability, interoperability, reusability and adaptability of DW systems.

In this paper, we focus on the MD modeling of the DW within our MDA framework. We have to develop a PIM (see figure 1) to specify every MD property without taking into account any technology. We consider a PIM as a MD

conceptual model, since it is independent of any database technology or DBMS. Then, a PSM is derived taking into account the platform in which the DW will be deployed. The PSM corresponds to a logical representation of the conceptual model, since it is related to a specific database technology (relational or multidimensional). Transformations between them are also formally established by using the QVT language.
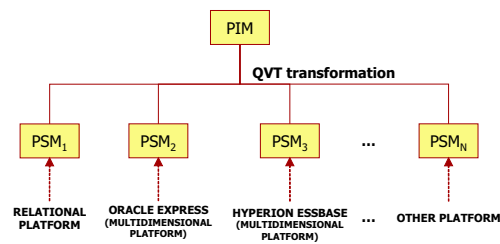

Fig. 1. Overview of our MDA approach for MD modeling of DW

In figure 1, we show a symbolic diagram that will help to understand our approach for the MD modeling of DWs:

- On the top of this figure we have represented the PIM. It corresponds with the MD conceptual schema, since it does not contain any technology-specific issue. Therefore, this PIM is modeled by using our UML profile for the MD modeling of DWs [7].
- From this PIM, we can develop several logical models (i.e. PSMs) using a relational or a multidimensional approach. These models are represented in the middle of figure 1. CWM [16] is used to specify every PSM, since CWM provides several metamodels to represent data structures according to certain technologies. From each PSM we can derive the necessary code to create data structures for DW in the platform which indicates the PSM. However, since PSM is defined with CWM, it is close to a specific technology, then it is quite straightforward to derive the corresponding code [20]. Thus, we do not develop this issue in this paper, and we focus on the transformation between PIM and PSM.
- On the bottom of the figure we represent that each PSM is related to one specific technology. As relational database technology is broadly extended, every DBMS supports all relational elements as tables, primary keys, and so on. Therefore, we only have to derive one PSM related to a relational approach. However, multidimensional databases technology depends on a specific DBMS since there is no standard to represents multidimensional elements and proprietary data structures are normally used. Thereby, several transformations are given in order to obtain the required PSM according to a specific tool (e.g. Oracle Express, Hyperion Essbase, and so on).

– Finally, transformations between PIM and every PSM are formally established by using the QVT language.

The main advantage of our approach is that the logical representation (i.e. PSM) of the DW is automatically generated once the PIM (conceptual model) is designed. Therefore the productivity is improved and development time and cost decrease. Other benefits of applying MDA to the MD modeling of DWs are the following:

– Since transformations represent repeatable design decisions, once a transformation is developed, we can use it in every instance of PIMs to generate the different PSMs for several projects. Therefore, we can include MD modeling best practices in MDA transformations and reuse them in every project to assure high quality DWs.
– If a new database technology arises, we do not have to change the PIM. Since it is platform independent, it is still valid, and only the transformations have to be updated in order to obtain the right PSM.
– As transformations are developed in a standard language like QVT, they can be adapted by designers according to their individual needs, since these transformations are not simple informal guidelines implemented ad-hoc on a concrete tool [20].

The following subsections deeply explain our PIM, PSM, and QVT transformations between them for MD modeling of DWs[2].

4.1 PIM for Multidimensional Modeling

A PIM is a view of a system from the platform independent viewpoint [18]. This means that this model describes the system hiding the necessary details related to a particular platform. This point of view corresponds with a conceptual level, i.e. representing the main MD properties without taking into account any specific technology detail. Therefore, the specification of the DW is independent from the platform in which it will be implemented. This PIM for MD modeling is developed following our UML profile presented in [7]. This profile contains the necessary stereotypes in order to carry out the conceptual MD modeling successfully (see table 1 and figure 2).

Specifically, the structural properties of MD modeling are represented by means of a UML class diagram in which the information is clearly organized into facts and dimensions. These facts and dimensions are represented by *Fact* and *Dimension* classes respectively. *Fact* classes are defined as composite classes in shared aggregation relationships of n *Dimension* classes. The minimum cardinality in the role of the *Dimension* classes is 1 to indicate that every fact must always be related to all the dimensions. The many-to-many relationships between a fact and a specific dimension are specified by means of the cardinality

---

[2] How the PIM is constructed is out of the scope of this paper, but we refer reader to [21] for a detailed explanation

Table 1. Main stereotypes of our UML profile for MD modeling of DWs

Stereotype	Description	Icon
Fact	Classes of this stereotype represent facts in a MD model, consisting of measures (the values being analyzed).	
Dimension	Classes of this stereotype represent dimensions in a MD model, consisting on hierarchy levels.	
Base	Classes with this stereotype represent dimension hierarchy levels in a MD model consisting of dimension attributes (descriptive information about the values being analyzed).	**B**
DegenerateDimension	Attributes of this stereotype represent degenerate dimension in a MD model.	**DD**
FactAttribute	Attributes of this stereotype represent attributes of a fact (i.e. measures) in a MD model.	**FA**
DimensionAttribute	Attributes of this stereotype represent attributes of a dimension hierarchy level (i.e. base) in a MD model.	**DA**
Descriptor	Attributes of this stereotype represent descriptor attributes of a dimension hierarchy level (i.e. base) in a MD model.	**D**
Rolls-UpTo	Associations of this stereotype represent relationships between two levels (i.e. bases) of a classification hierarchy in a MD model.	`<<Rolls-UpTo>>`
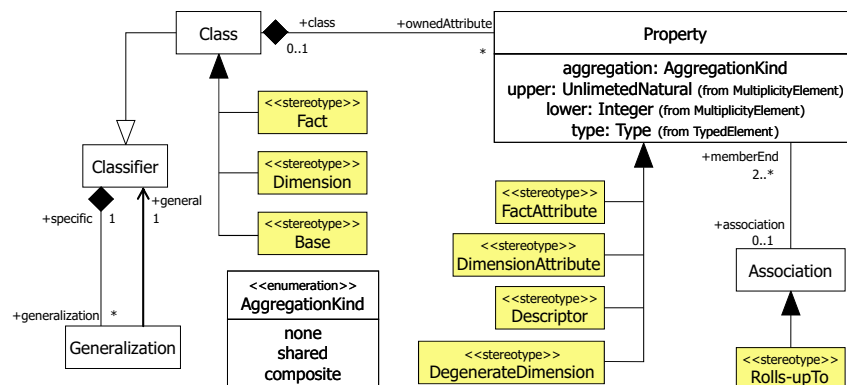
Fig. 2. Part of the UML metamodel extended by our profile

1...n in the role of the corresponding *Dimension* class. A fact is composed of measures or fact attributes. These are represented as properties with the *FactAttribute* stereotype. By default, all measures in the *Fact* class are considered to be additive. For non-additive measures, additive rules are defined as constraints and are included in the fact class. Furthermore, derived measures can also be explicitly represented (indicated by /) and their derivation rule is considered as a tagged value of a *FactAttribute*.

Our approach also allows the definition of degenerated dimensions [1], thereby representing other fact features in addition to the measures for analysis. These degenerated dimensions are represented as stereotyped properties of the *Fact* class (*DegenerateDimension* stereotype).

With respect to dimensions, each level of a classification hierarchy is specified by a *Base* class. Every *Base* class can contain several dimension attributes (*DimensionAttribute* stereotype) and must also contain a *Descriptor* attribute (stereotype *D*). An association (represented by a stereotype called *Rolls-UpTo*) between *Base* classes specifies the relationship between two levels of a classification hierarchy. The only prerequisite is that these classes must define a Directed Acyclic Graph (DAG) rooted in the *Dimension* class (DAG constraint is defined in the stereotype *Dimension*). The DAG structure can represent both multiple and alternative path hierarchies.

A *Dimension* class contains a unique first hierarchy (or dimension) level called terminal dimension level (or root level). A roll-up path is a subsequence of dimension levels, which starts in a root level (lower level of detail) and ends in an implicit level (not graphically represented) that represents all the dimension levels.

We use roles to represent the way the two classes see each other in a *Rolls-UpTo* association: role *R* represents the direction in which the hierarchy rolls-up, whereas role *D* represents the direction in which the hierarchy drills-down. Moreover, we use roles to detect and avoid cycles in a classification hierarchy, and therefore, help us to achieve the DAG condition.

Due to flexibility of UML, we can also consider non-strict hierarchies (an object at a hierarchy's lower level belongs to more than one higher-level object) and complete hierarchies (all members belong to one higher-class object and that object consists of those members only). These characteristics are specified, respectively, by means of the cardinality of the roles of the associations and defining the stereotype *Completeness* in the association between *Base* classes. Lastly, the categorization of dimensions is considered by means of the generalization/specialization relationships of UML.

Our profile is formally defined and uses the Object Constraint Language (OCL) [22] for expressing well-formed rules of the new defined elements (see table 1), thereby avoiding an arbitrary use of the profile. We refer reader to [7] for a further explanation of this profile and its corresponding OCL constraints.

4.2 PSM for MD Modeling

A PSM is a view of a system from the platform specific viewpoint [18]. It represents the model of the same system specified by the PIM but it also specifies how that system makes use of the chosen platform or technology. In MD modeling, platform specific means that the PSM is specially designed for a kind of a specific DBMS. These systems can use relational technology (relational database to store multidimensional data) or multidimensional technology (structures the data directly in multidimensional structures).

In our approach, each PSM is modeled by using the resource layer from CWM (Common Warehouse Metamodel) [16], since it is a standard to represent the structure of data. CWM metamodels can all be used as source or target for MDA transformations, since they are MOF-compliant. Specifically, we use the relational and the multidimensional metamodels:

- **Relational metamodel.** It contains classes and associations that represent every aspect of relational databases. With this metamodel we can represent tables, columns, primary keys, foreign keys and so on. In a previous work [8] we have used this metamodel to obtain a relational PSM.
- **Multidimensional metamodel.** It contains common data structures that represent every MD property. However, multidimensional databases are not as standardized as relational ones, since the former generally defines proprietary data structures. Therefore, this multidimensional metamodel only define commonly used data structures in order to be enough generic to support a vendor specific extension. In this paper, we use the part of the multidimensional metamodel shown in figure 3 that correspond to an Oracle Express extension defined in Volume 2, Extensions, of the CWM Specification [23].
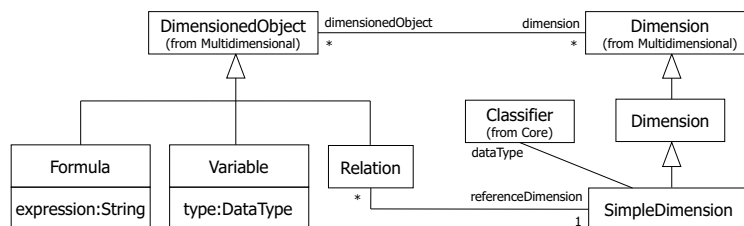
Fig. 3. Multidimensional metamodel for Oracle Express

4.3 QVT Transformations for MD Modeling

Developing formal and automatic transformations between models (e.g. between PIM and PSM) is one of the strong points of MDA [20]. In this paper, transformations are given following the declarative approach of QVT [19]. Therefore,

according to the QVT relations language, we have developed every relation to obtain a transformation[3] between our PIM and a PSM for a multidimensional platform (Oracle Express). Every relation is shown in figures 4-10. Due to space constraints we only focus on one of them: `Dimension2SimpleDimension` (see figure 4). On the left hand side of this relation we can see the source model and on the right side the target model. The source model is the part of the PIM metamodel (see figure 2) that has to match with the part of the PSM meta-model (see figure 3) which represents the target model. In this case a collection of elements, that represents a *Dimension* class together with the root *Base* class (i.e. terminal dimension level) of our UML profile for MD modeling, matches with a *SimpleDimension* class from the CWM multidimensional package. This element must have the same name that the *Dimension* class of the source model and its type is obtained from the function `UML2OEType` (defined in figure 11). This function turns a UML data type into a Orale Express type.

`Dimension2SimpleDimension` relation determines the transformation in the following way: it is checked (C arrow) that the pattern on the left side (source model) exist in the PIM, then the transformation enforces (E arrow) that a new *SimpleDimension* class, according to the PSM metamodel, is created with its cor-responding name and type. Once this relation holds, the following relations must be carried out (according to the *where* clause): `DimensionAttribute2Variable`, `Base2SimpleDimension`, and `Fact2SimpleDimension`.
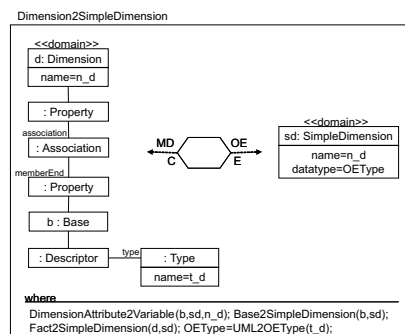
Fig. 4. Relation for Dimension
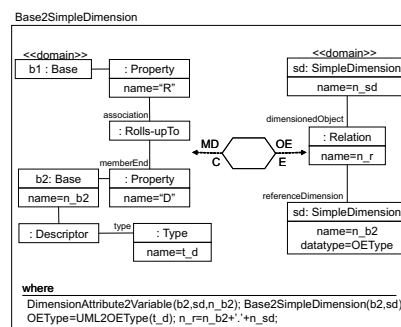
Fig. 5. Relation for Base

5 Example

In this section, an example is introduced in order to better show how to apply MDA and QVT transformation rules to the MD modeling of the DWs. This ex-

---

[3] Due to space constraints, we focus on the main MD properties represented by our profile, thus QVT transformation rules including other features like non-strict or complete hierarchies, additivity rules or categorization of dimensions are not shown in this paper.
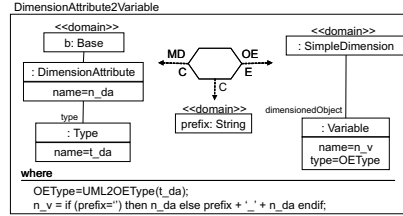
Fig. 6. Relation for DimensionAttribute
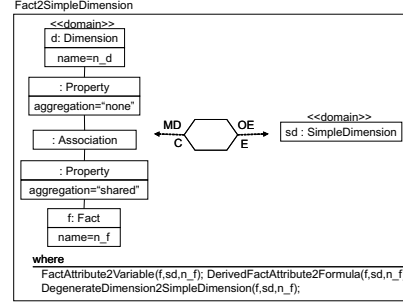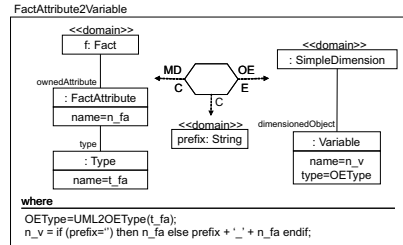

Fig. 7. Relation for Fact

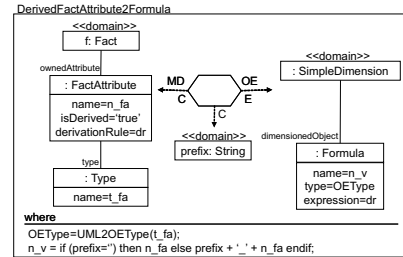**Fig. 8.** Relation for FactAttribute

Fig. 9. Relation for derived FactAttribute

ample is inspired by a case study from [24]. It relates to a company that comprises different dealerships that sell automobiles across several countries. Therefore, we focus on the *automobile sales* fact. This fact contain several measures (i.e. fact attributes) to be analyzed (*quantity*, *price* and *total*). Furthermore, we specify a number of contract (*contractN*) as a degenerate dimension. On the other hand, we also consider the following dimensions as contexts to analyze measures: *time*, *product*, and *customer*. However, we focus on the *customer* dimension, which have the hierarchy levels: *customer personal data*, *city*, *region*, and *country*. The corresponding PIM is specified in figure 12.

From the defined PIM, we can apply the transformation above-defined in order to obtain the corresponding PSM. An example can be viewed in figure 13 where two relations of the transformation are applied: `Base2SimpleDimension`, and `DimensionAttribute2Variable`. In this figure we can see that every level of a dimension hierarchy (*Base* classes) becomes into a single dimension, and the association between levels in the PIM (*Rolls-upTo* associations) corresponds to a *Relation* in the PSM. Furthermore, *DimensionAttributes* from PSM are converted into *Variables* in the PSM.

The resulting PSM after applying every relation described in section 4.3 to defined PIM (see figure 12) is shown in figure 14.
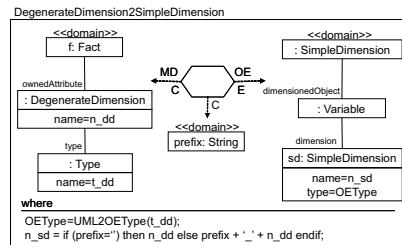
Fig. 10. Relation for DegenerateDimension

Fig. 11. Converting types

6 Conclusion and future work

In this paper, we have presented an MDA approach for the MD modeling of DWs. We have focused on defining a PIM for MD modeling, a PSM according to a multidimensional database technology by using CWM, and a set of QVT transformation rules in order to derive the PSM from the PIM. An example of applying our approach has been given in order to show every of the developed QVT transformations.

According to our MDA framework for the development of the DW [8], we plan to develop transformations for each part of a DW system by using other CWM metamodels. Moreover, we plan to enrich the presented transformations by adding metrics in order to be able to obtain the highest quality PSM.

7 Acknowledgements

References

1. Kimball, R., Ross, M.: The Data Warehouse Toolkit (2nd Edition). John Wiley & Sons (2002)
2. Inmon, W.: Building the Data Warehouse (3rd Edition). Wiley & Sons (2002)
3. Trujillo, J., Palomar, M., Gómez, J., Song, I.Y.: Designing data warehouses with OO conceptual models. IEEE Computer **34**(12) (2001) 66–75
4. Golfarelli, M., Rizzi, S.: Methodological framework for data warehouse design. In: DOLAP, ACM (1998) 3–9
5. Tryfona, N., Busborg, F., Christiansen, J.G.B.: starER: A conceptual model for data warehouse design. In: DOLAP, ACM (1999) 3–8
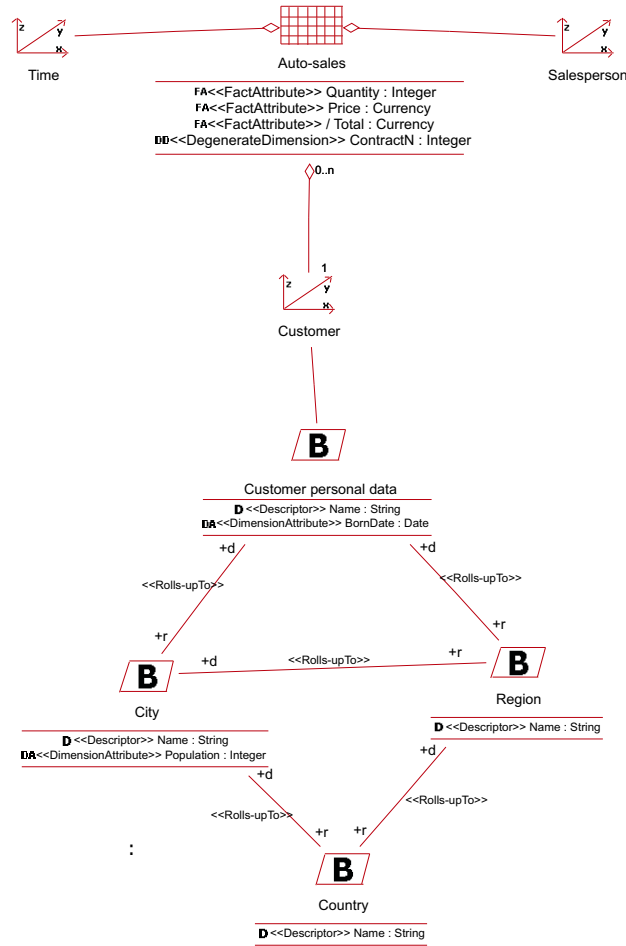
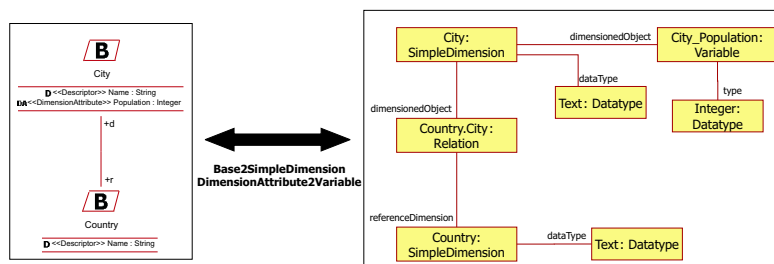**Fig. 12.** Example of a DW for auto sales (PIM)
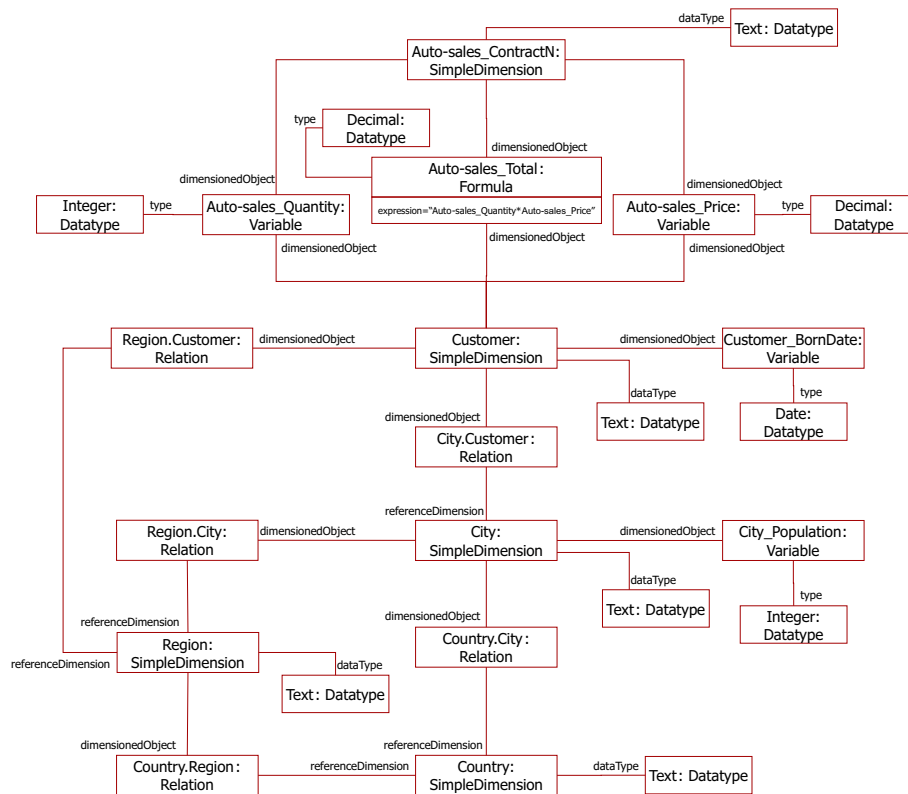

Fig. 13. Applying two QVT relations

Fig. 14. Oracle Express implementation for auto sales example (PSM)

6. Pedersen, T.B., Jensen, C.S.: Multidimensional database technology. IEEE Computer **34**(12) (2001) 40–46
7. Luján-Mora, S., Trujillo, J., Song, I.Y.: A UML profile for multidimensional modeling in data warehouses. Data & Knowledge Engineering (**In Press**)
8. Mazón, J.N., Trujillo, J., Serrano, M., Piattini, M.: Applying MDA to the development of data warehouses. In: DOLAP. (2005) 57–66
9. Abelló, A., Samos, J., Saltor, F.: A framework for the classification and description of multidimensional data models. In Mayr, H.C., Lazanský, J., Quirchmayr, G., Vogel, P., eds.: DEXA. Volume 2113 of Lecture Notes in Computer Science., Springer (2001) 668–677
10. Bézivin, J., Hammoudi, S., Lopes, D., Jouault, F.: Applying MDA approach for web service platform. In: EDOC, IEEE Computer Society (2004) 58–70
11. Meliá, S., Gómez, J., Koch, N.: Improving web design methods with architecture modeling. In Bauknecht, K., Pröll, B., Werthner, H., eds.: EC-Web. Volume 3590 of Lecture Notes in Computer Science., Springer (2005) 53–64
12. Vanderdonckt, J.: A MDA-compliant environment for developing user interfaces of information systems. In Pastor, O., e Cunha, J.F., eds.: CAiSE. Volume 3520 of Lecture Notes in Computer Science., Springer (2005) 16–31
13. Burmester, S., Giese, H., Schäfer, W.: Model-driven architecture for hard real-time systems: From platform independent models to code. In: ECMDA-FA, Lecture Notes in Computer Science, Vol. 3748. (2005) 25–40
14. Maria, B.A.D., da Silva, V.T., de Lucena, C.J.P.: Developing multi-agent systems based on MDA. In Belo, O., Eder, J., e Cunha, J.F., Pastor, O., eds.: CAiSE Short Paper Proceedings. Volume 161 of CEUR Workshop Proceedings. (2005)
15. Poole, J.: Model Driven Data Warehousing MDDW. `http://www.cwmforum.org/POOLEIntegrate2003.pdf`. (Visited January 2006)
16. Object Management Group: Common warehouse metamodel (CWM) Specification 1.1. `http://www.omg.org/cgi-bin/doc?formal/03-03-02` (Visited January 2006)
17. Medina, E., Trujillo, J.: A standard for representing multidimensional properties: The Common Warehouse Metamodel. In Manolopoulos, Y., Návrat, P., eds.: ADBIS. Volume 2435 of Lecture Notes in Computer Science., Springer (2002) 232–247
18. Object Management Group: MDA Guide 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01` (Visited January 2006)
19. Object Management Group: MOF 2.0 Query/View/Transformation. `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01` (Visited January 2006)
20. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. The Practice and Promise of The Model Driven Architecture. Addison Wesley (2003)
21. Mazón, J.N., Trujillo, J., Serrano, M., Piattini, M.: Designing data warehouses: from business requirement analysis to multidimensional modeling. In Cox, K., Dubois, E., Pigneur, Y., Bleistein, S.J., Verner, J., Davis, A.M., Wieringa, R., eds.: REBNITA, University of New South Wales Press (2005) 44–53
22. Object Management Group: Unified Modeling Language Specification 2.0. `http://www.omg.org/cgi-bin/doc?formal/05-07-04` (Visited January 2006)
23. Object Management Group: Common Warehouse Metamodel (CWM) Specification 1.1. Volume 2. Extensions. `http://www.omg.org/cgi-bin/doc?ad/2001-02-02` (Visited January 2006)
24. Giovinazzo, W.: Object-Oriented Data Warehouse Design. Building a Star Schema. Prentice-Hall (2000)

UPT: A Graphical Transformation Language based on a UML Profile[1]

Santiago Meliá, Jaime Gómez, and Jose Luís Serrano

Universidad de Alicante, Spain
{santi,jgomez,monfor}@dlsi.ua.es

Abstract. Model transformations are recognised as being of crucial importance for the successful execution of Model Driven development proposals. These artefacts are especially important in the software engineering community where design methods need to transform their models into software implementations. Particular attention has recently been paid to model transformations defined by the OMG standard Queries/Views/Transformations (QVT). However, QVT only provides us with a theoretical help because it does not have a complete tool support yet. In this work, we present an MDA approach called UPT (UML Profile for Transformations). UPT defines a new UML profile which allows us to represent metamodel transformations using any UML tool and send these transformations to the different tools using the XMI standard.

Introduction

A new paradigm has appeared in the last few years called Model-Driven Engineering (MDE) [7] which makes intensive use of models for the representation of different views of the system. The key concept behind MDE is that all artefacts generated during the development of software should be represented by common modelling languages. Consequently, software development can be seen as the process of transforming a model into another until it can be executed outside its development environment. Unlike previous development approaches, MDE proposes the use of the concept of transformation. Transformation is the really new element in the MDE, because it makes possible to formalize the evolution of a model into another or even to add new characteristics to that same model. However, MDE is still young and the first tools have not provided the desired solutions yet. Particular attention has recently been paid to the model transformations defined by the OMG standard Query/Views/Transformations (QVT) [14]. But as QVT is not sufficiently integrated in the current tools that represent standards such as UML [16], OCL [15] or XMI [17], it is hard to obtain a tool able to support it.

In this paper, we present a new approach called UPT (UML Profile for Transformations) which represents metamodel transformations using any UML tool and transports them using the XMI [17] standard.

In order to understand the motivation of our approach, we should first give an overview of the problems currently facing the QVT Relation specification (see this in section 2). Our description of the UPT approach starts with Section 3. Section 3.1 presents a description of the UPT elements; Section 3.2 proposes the formalization of the UPT model by means of the UPT metamodel; Section 3.3 presents the UPT standardization with the UML Profile and section 3.4 gives an example of UPT transformation based on a well-known mapping between a UML Class and a RBMDS Table. In section 4, a brief description of the Web UPT tool is presented. Finally, the relevant related work and the future lines of research are outlined in section 5 and 6, respectively.

The QVT-Merge Approach

OMG has recognized that Model transformations are crucial for the success of the MDE. Hence, the Query/Views/Transformations (QVT) RFP[11] to seek an answer compatible with its MDA standard suite: UML, MOF, OCL, etc. Several formal replies were given by a number of research institutions and these evolved during the last three years into a single proposal [14] given by QVT-Merge group consisting in a large number of original submitters. The QVT-Merge specification has a hybrid declarative/imperative nature, with the declarative part being split into a two-level Architecture: (1) A user-friendly Relations language which supports complex object pattern matching and an object template creation. The relation language can be represented by a graphical and textual notation. (2) A Core metamodel which has defined a language using minimal extensions to EMOF and OCL.

Following the MDE philosophy, where the model is the most important artifact of the development process, we consider that the majority of transformations should be represented by the QVT graphical notation. However, QVT is still very young and has not developed a tool that can satisfy it completely yet. The main reasons are:

- The QVT Relation language has a new graphical notation which is not compliant with the UML notation. In fact, up to now there is no adequate tool able to support the QVT graphical notation.
- The graphical language represents the object patterns using the UML object model. However, this object model has to represent different parts of the metamodel which is a class model. For this reason, the graphical language does not have the sufficient expressiveness which would allow it to represent the different parts of the metamodel. For instance, the cardinality of the roles associations is not represented in the object pattern and in some cases, for example if we want to navigate it is necessary to know whether the opposite role cardinality is 0:1 or 0:M.
- The graphical and textual language requires a rather long learning process before it can be used. The transformation developer must change his way of thinking, that is, he has to represent the instances of metaclasses as objects.

- The QVT relation transformations have low tool compatibility which means that QVT transformations defined in a tool cannot be read in another tool.

Next section describes the UML profile for Transformations which resolves the problems of the QVT graphical notation.

UML Profile for Transformations (UPT)

A profound study on declarative transformations defined by QVT detected certain semantic and graphic flaws. These problems have induced us to define a new transformation language inspired by QVT standard that tries to solve these problems and to improve its graphical and semantic aspects.

The result is a solution for the definition of transformations which we have called UPT (UML Profile Transformation) which has introduced a new graphical notation based on the UML Class Diagram which facilitates the specification of metamodel transformations. UML and more specifically the class diagram, as was recognised by empiric works such as [4], it represents the best known and tested form of model notation. Also, UPT presents a small MOF metamodel establishing the necessary elements for its specification so as to be able to define it as a UML profile.

The UPT main advantages are: (1) it can be expressed by a great number of tools able to represent UML models graphically. It reduces the cost of having to define new tools because we use UML tools to represent transformations. (2) UML allows us to send transformation models to different tools by means of the standard XMI. (3) UML will allow us to introduce new concepts and new semantics to represent transformations through profile mechanisms. (4) The class diagram is the model most used by analysts in UML. It provides a smaller learning curve to modellers who want to define transformations using UPT.

In conclusion, we believe that UPT is a rich and standard graphical transformation language and does not present the shortcomings of the QVT Relation language.

Next, we will focus on the definition of the different elements that represent the UPT transformations.

The UPT Elements

In order to represent the transformations through the UPT proposal, we have to describe the elements that make it up. The elements defined by UPT are initially inspired by those given by QVT. However, UPT has introduced extensions that improve the graphical notation reducing the ambiguities based on the class diagram. This gives UPT the possibility to tackle the metamodels in a more precise and clear way than did the objects diagram used by QVT.

We will follow a top-down description, that is, we will go from general to more precise concepts.

Transformation

A specific transformation is a set of relations or transformation rules that have to be carried out within the elements of a set of candidate models. Each candidate model and the types of its elements are restricted to a metamodel. We shall call this candidate model "typed model". This way, the transformation contains a reference to the different typed models that take part in it.

In QVT and UPT, the transformations are defined at the level of metamodel, that is, they specify the relations between metaclases of a set of origin metamodels and the metaclases of a set of target metamodels. However, their application is at a model level and these definitions allow them to be applied to any type of model that fulfils the defined rules of transformation.

To represent a transformation in UPT, we will use the element UML Package. This container element will include the set of transformation rules or relations that must be applied.

Relation

The Relation is the central element in the UPT declarative specification. A relation indicates the restrictions that the elements of the model candidates must fulfil. Each relation is made up of two or more domains and two restrictions known as when (or guard) and the clause where. Each domain designates a model candidate, and in more simple cases, a type to match this model.

The relation in UPT is represented by a UML class model, where we define a stereotyped class by the name of <<Relation>>. This class will contain a series of properties and relations associated to this relation (see Fig. 1).

Each relation has the following properties:

- **IsRoot**: is represented as a boolean type attribute. Its value is true whether the relation is invoked from the transformation and not from a where belonging to another relation. Its value is false if the relation is invoked from the where of another relation. Each transformation must have one or more top relations which will mark the start of its execution.

- **When**: is represented as a restriction of the class <<Relation>>. The clause When specifies an OCL condition that must be accomplished by the variables of the relation in order to be carried out. It consists of an OCL predicate, which must be executed before the relation.

- **Where**: is represented through a restriction of the class <<Relation>>. The clause where specifies a condition that must be fulfilled by all the elements that have taken part in the relation. It consists of an OCL predicate, normally formed by a set of calls to other relations. The notation used to represent the calls consists in a set of calls separated by the symbol ';'. The where is executed once the relation has been completed.

In addition to the properties, each class <<Relation>> contains a series of association relationships with the different domains that participate in it.

Domain

A domain specifies the set of elements of a typed model that are of interest for a relation. A domain within the UPT profile will be represented by a relation of

association between the class relation and by the DomainPattern class (the DomainPattern class is stereotyped by the label <<Domain>> see Fig. 1).

Each association relationship between the relation class and the DomainPattern class contains the following properties associated to the domain:

- **Typed Model**: is represented by the name of the association relationship. The typed model indicates the part of the metamodel which takes part in the relation. In Fig. 1 we can see how the typed model is specified by namespaces *metamodelOrigin* and *metamodelTarget* in which we navigate from the outside into the inside, up to the concrete package of the metamodel where we are going to require the domain elements.

- **Checkable or Enforceable**: This property is represented through the stereotype of the association relationship. It can have the values <<C>> if the domain is checkonly and <<E>> if the domain is enforceable. If it is a checkonly domain, it indicates that we are going to make sure that the elements defined in the domain are to be found in the same way in the element of reference. If we find that they do not coincide then the relation fails. If, on the other hand we find that the domain has the label <<E>> (enforceable) we are indicating that if the domain elements are not the same in the reference model we must modify them until we succeed. In order to do that we can create, cancel or modify elements in the reference model. Normally, this domain corresponds to the destination model, while domains that have the label <<C>> correspond to the origin models.

- **RootVariable**: in UPT the domain is also represented by a stereotyped class <<Domain>> that we call the class RelationDomain. The name of this class is called RootVariable and it represents the element of the model that marks the entry point on the typed model.

Fig. 1 represents the notation of the UPT Relation. The class <<Relation>> is called *R*. This class <<Relation>> contains the tagged value *isRoot* equals true, indicating that the relation is invoked when the transformation starts. Besides, it contains two associations with different DomainPattern elements (stereotyped classes with the label <<Domain>>). Each association (which represents a Domain) has its path to the metamodel to which each domain belongs (typed model). In this case, we refer to them as *OriginMetamodel* and *TargetMetamodel*.

The first domain is type <<C>> where the relation must confirm that the elements of the *OriginMetamodel* are within this domain. In this case, The RootVariable is *D1* and has a PropertyPattern called *roleO1* which represents a set of instances of *Metaclass1*. If that is not the case the relation is not accomplished. Only if we fulfill the checkonly domain, we can pass to the phase of executing the enforceable domain. The enforceable domain is represented by the association stereotyped by <<E>> which will force the existence of the *TargetMetamodel* elements in the domain. If this was not the case, it would modify the model until it complied with the domain. The *RootVariable* is D2 and has a PropertyPattern called *roleO2* which represents a set of instances of *Metaclass2*.
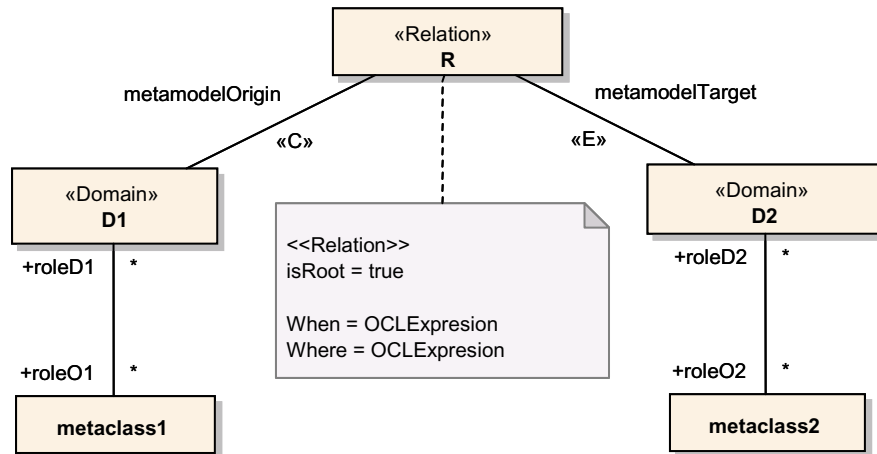
Fig. 1. **Notation of the UPT Relation**

DomainPattern

Each domain is associated with a DomainPattern which can be seen as a template class, with its properties and relations. In order to fulfill the relation these have to be found, modified or created. A DomainPattern specifies a template class in the form of an arbitrarily complex graphic, where the classes belong to different metaclasses of the domain reference metamodel. This is the reason why the template defined by the DomainPattern starts with the same rootVariable defined in the domain. This provides the metaclass from which we start to define the pattern.

Within a DomainPattern we find the following elements:

- **ClassPattern**: it receives this name because it refers to a metaclass instance, that is, a class. In UPT is represented by a class. If we review the metamodel architecture of OMG [12], we will see that the instance of a metaclass is a class. For this reason, the class is its most natural representation (unlike QVT where an object is used to represent an instance of a metaclass by leaping over 2 layers of the metamodel architecture). An ClassPattern can contain at the same time a set of PropertyPattern elements.

- **PropertyPattern**: it specifies restrictions in the values that the ClassPattern can take. There are two types: (1) **attributes** belonging to the class defined by a ClassPattern, it is specified in the same way as an attribute of a UML class. It is composed by the name, a type and the value taken by that attribute. The value can be a variable if it is specified without "quotation marks" and this tells the relation that attribute can take any value within the range of the type, or it can come between "quotation marks" which indicates that it is a "literal", fixing the value of its attribute. In UPT we have added the type, (unlike the QVT specification) this allows us to restrict the range of values taken by the attribute. Finally we would like to indicate that one or more attributes can be stereotyped as <<key>>, this allows us to indicate that attributes identify in a unique way each instance of the referred metaclass. If the key attribute belongs to a ClassPattern contained in a composition (or part), its identity will be unique for each instance

of the composed ClassPattern (or whole). (2) **Role** or element association relationships between different ClassPatterns. This appears when the PropertyPattern is the end of an association, in which case the type of the PropertyPattern will be the type of the ClassPattern that the association has at the end. It possesses the **Cardinality**: specified by two natural numbers which indicates the minor or major number of instances that contain this role (1, 0..., 0...n, 1...n. etc.). The fact that we can put a minor cardinality equal to zero in the role of a ClassPattern indicates that it is an optional part of specified by the Relation. This introduces a richer semantic than the representation used by the objects diagram of the QVT, because the relationships between instances are always mandatory. Besides, the role possesses the property of **isComposition**, indicating when this PropertyPattern is contained by a composition in another PropertyPattern, that is, is responsible of the existence (it affects its identity) and the storage of the same.

- **AssociationPattern:** it establishes a relationship between the instances of two ClassPatterns. It allows to reproduce the association relationship established in the metamodel referred to by the domain between the instances of two of its metaclasses. But with two different semantics: (1) if the domain is checkonly it means that the instances fulfil this relation, (2) if it is enforceable we create, cancel or modify the necessary instances so that the AssociationPattern can be compliant with the association.

Metamodel

In the UPT metamodel we formalize the elements and the possible relations that make it possible to represent graphically transformations and relations.

In an effort of simplification we have been able to make this metamodel compatible with the UML metamodel. Besides, due to the nature of the description of relations based on classes instead of on objects, the UPT metamodel possesses some advantages over the QVT metamodel, such as the elimination of added concepts (collections of objects due to the use of cardinality, the use of CompositionPattern, the definition of a dependency in the order of execution between transformations, etc.)

The aim of this section is to describe the UPT metamodel, focusing mainly on the new characteristics introduced by our approach.

Fig. 2 shows the packet Metamodel Transformation. Its main concept is the metaclass Transformation. Its recurrent relationship allows us to establish the inheritance relationship between different transformations.

On the one hand, we have defined the class DependencyTrans which allows us to establish an order of execution (successor and predecessor) between the different transformations; this characteristic will be very useful when we want to define the map of transformations.

At the same time a Transformation contains a set of instances of the class Rule (abstract class which overrides Domain).

Each rule defines its name and a recursive Relation that allows for the rule to be overridden .Besides, a Rule is composed by a set of n instances of the Domain metaclass.

This association relationship allows for the rules to be n:m, that is, n origin domains and m destination domains (where n and m are more or equal to 0).

We should finally indicate that each instance of the Domain class possesses two attributes: (1) **isCheckable** which specifies if we are just going to check the metamodel (checkable equals to true) and (2) type String Metamodel, indicates the path of the metalmodel to which we refer to.
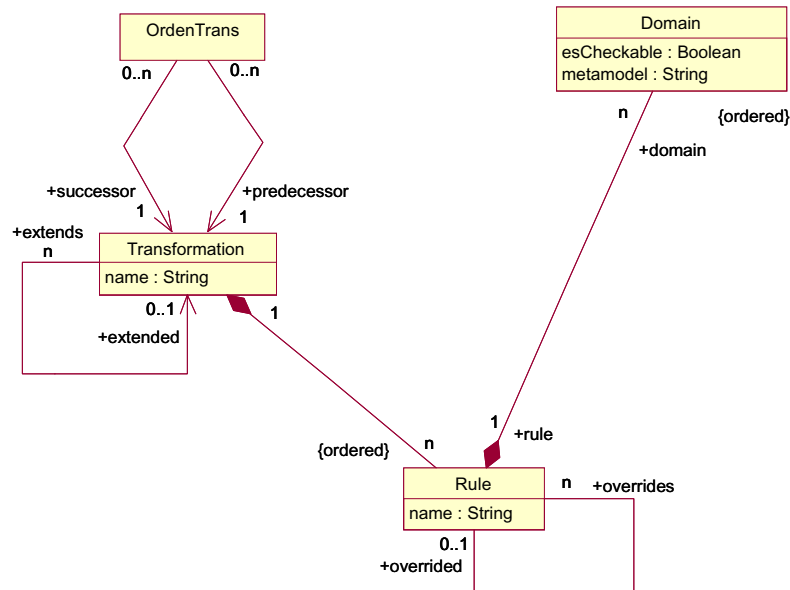

Fig. 2.   Transformation Package of the UPT Metamodel

The next UPT metamodel package is called Relations (see Fig. 3). Its main element is the Relation which is the main unit of the UPT declarative specification.

Fig. 3 shows that the *Relation* metaclass is an extension of *Rule* and inherits its properties and the link with the *Domain* metaclass. Each *Relation* metaclass contains an *isRoot* attribute which indicates whether this Relation is or is not a root, that is, if its value is true the Relation is directly invoked by the Transformation while if it is false it is invoked from the Where of another Relation.

Besides, it contains two relations with the *Pattern* metaclass which correspond to the OCL expressions *Where* and *When*. Each Pattern metaclass, in turn, is made up of a set of *Predicate* elements which contains an *OCLExpression*.

Finally, the *OCLExpression* contains a reference to the instances of the *Variable* metaclass where the name and type of each Variable used by the OCLExpression is stored in the metamodel.

On the right side, the *Domain* metaclass is related to a *ClassPattern* called *RootVariable* which represents the instance of the metaclass which marks the entry point to the referred metamodel. Each *ClassPattern* has two attributes: (1) *class* which is a String which indicates the reference metaclass and (2) *variable* which has the type of the *Variable* metaclass. We should also indicate that the *ClassPattern* contains its

own attributes, a set of *PropertyPattern* elements which allows us to introduce both simple type properties and associations properties through an association with another *ClassPattern* instance. For simple properties, the value is an *OCLexpression* and if the property *isKey*, that is, if it forms a part of the *ClassPattern* identification.
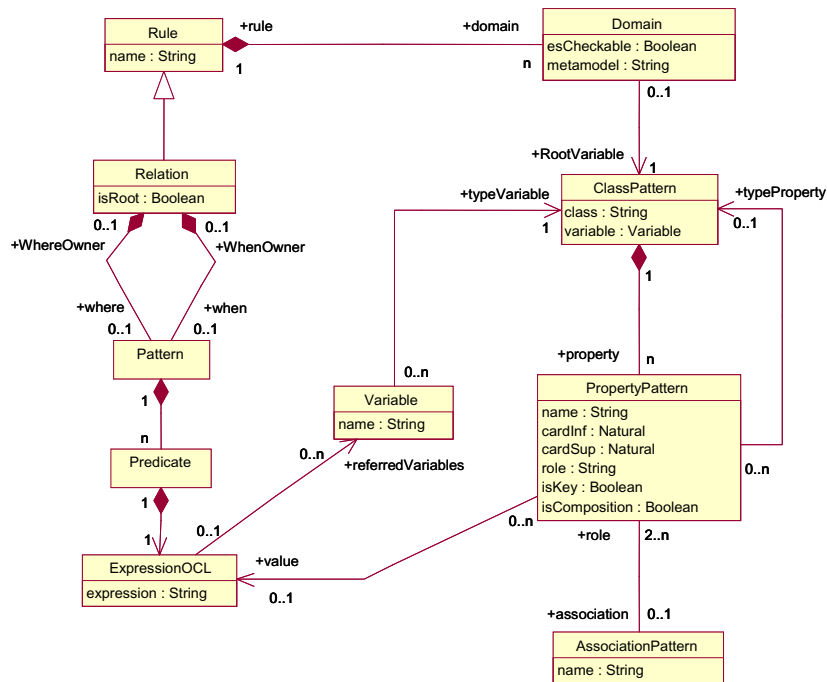

Fig. 3.   Relation Package of UPT Metamodel

Furthermore, the *PropertyPattern* is a role of an association. Here, we have to highlight the UPT modification where we have introduced the *AssociationPattern* metaclass. It allows us to establish association relations between metaclasses, unlike the QVT where the relations are between objects. UPT incorporates to the association relations the information of the role name, its inferior and superior cardinality (*cardInf* and *cardSup*) and it is indicated whether the association is or is not a composition. (*isComposition*).

The Definition of the UPT Profile

The UPT profile is the adaptation of each one of the concepts defined by the metamodel of our proposal to the standard UML metamodel [16]. Besides, as we need to query the models in the UPT transformations we have to use the OCL in its specification. To do this, the UPT has followed the alignment proposed by the OCL specification [15], which allows us to insert the different OCL expressions in the UML. As indicated previously, UPT has chosen the class diagram to express the Patterns defined by the transformations. This means that the profile must look for the

adjustment of each one of its concepts with the metaclasses defined in UML for the class diagrams and in OCL for its queries.
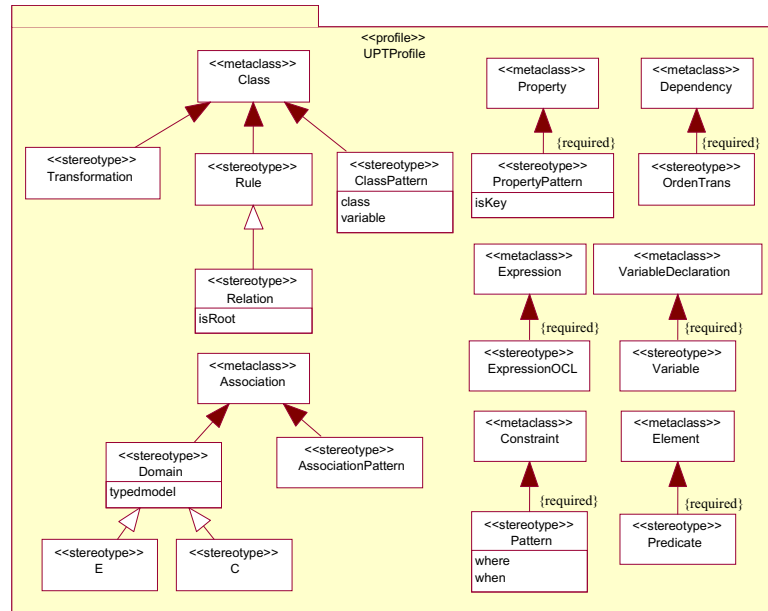

Fig. 4.   The UML Profile of Transformations (UPT)

In Fig. 4 we can see the link between the different stereotypes and the metaclasses on which the extension is carried out. The description starts from the Class metaclass, from which the stereotypes Transformation, ClassPattern and Rule derive. It is worthwhile highlighting that the Rule stereotype is abstract and cannot be used directly. Its son, the stereotype Relation is instanced instead.

On the right hand side, we see how the stereotype PropertyPattern extends from Property, thus maintaining the same relation with ClassPattern as does Property with class. The stereotype *OrdenTrans*, which is used to establish order between transformations. It uses the metaclass UML Dependency for this functionality.

On the left lower part, we can see how Domain and PatternAssociation extends from Association metaclass. Both stereotypes are used to establish relations between Relations and ClassPatterns, respectively.

On the lower right side, the Fig. 4 presents the stereotypes used for the definition of expressions within the UPT. *OCLExpression* indicates where these expressions are going to be located and to do this it extends the metaclass Expression. As [16] indicates, the UML metaclass Expression is the correct place where an OCL expression should be located.

The stereotype Variable is also defined as a VariableDeclaration extension. This metaclass belongs to the OCL metamodel and allows us to declare variables and indicate what type they are. On the lower part, there are two stereotypes used for the definition of restrictions *Where* and *When*. On the one side the stereotype Pattern which is defined as an extension of the UML metaclass *Constraint*. *Constraint* is

another point of definition of OCL expressions that are linked to the scope of one class. In our case the OCL expressions are linked to a Relation.

Finally, the *Predicate* is defined as an extension of the *Element* metaclass and represents a Boolean OCL expression belonging to a Pattern.

Besides matching stereotypes with the UML and OCL metaclases, we must represent the UPT metamodel relations by means of the definition of restrictions on the extended metamodels.

In the next section, we make use of the UPT language to specify a UPT relation example within the QVT specification.

An Example of UPT Relation

This example is obtained from the additional examples of the QVT-Merge specification [14] where we want to show its compatibility with our approach. Fig. 5 presents a UPT relation that maps the persistent classes of a simple UML model to tables of a simple RDBMS model.

Thanks to the simplicity given by the UPT graphic notation, we can easily see how the *ClassToTable* relation queries whether one or more elements *Class* belonging to a *Package* exist in the checkonly *UML* domain. Also, this Relation forces the existence of a *Table* (will create, modify or delete) which gets its name from the *Class*. Besides, the *Table* belongs to a *Schema* and contains a set of *Column* and *Key* elements. In spite of its simplicity, this relation contains most of the previously defined elements of the UPT profile. We can see how the domains stereotyped as <<Domain>> correspond to two ClassPattern elements called *Class* and *Table*, respectively. Both elements share the attribute called *name* of the type String and value *cn*. This value is specified without "quotation marks", therefore, the name of the *Class* and *Table* can take any value within the range of the type String. Besides, each ClassPattern possesses AssociationPatterns. On the one hand, *Class* has a composition relationship with the ClassPattern *Package*. The cardinality of this composition relationship indicates that for each *Package* there are zero or more *Class* instances and for each *Class* there is one instance of *Package*. It is worthy of remark that this information cannot be expressed in the QVT notation.

On the other hand, the ClassPattern *Table* has two composition relationships with Column and Keys. For each *Table* there are zero or more *Column* instances and for each *Column* belongs to a *Table*. The composition between *Table* and *Key* has the same cardinality as the previous relationship.

Finally, we want to give an explanation for the information introduced into the *ClassToTable* Relation. First of all, the tagged value isRoot=false indicates that the *Relation* is invoked from other relations in its transformation. Secondly, the ClassToTable relation has two Constraints: (1) the clause When which invokes the *PackageToSchema* Relation and confirms that this relation is accomplished by *Package* and *Schema* ClassPatterns. (2) The clause Where which invokes the *AttributeToColumn* Relation and must be satisfied by the *Class* and *Table* ClassPatterns.
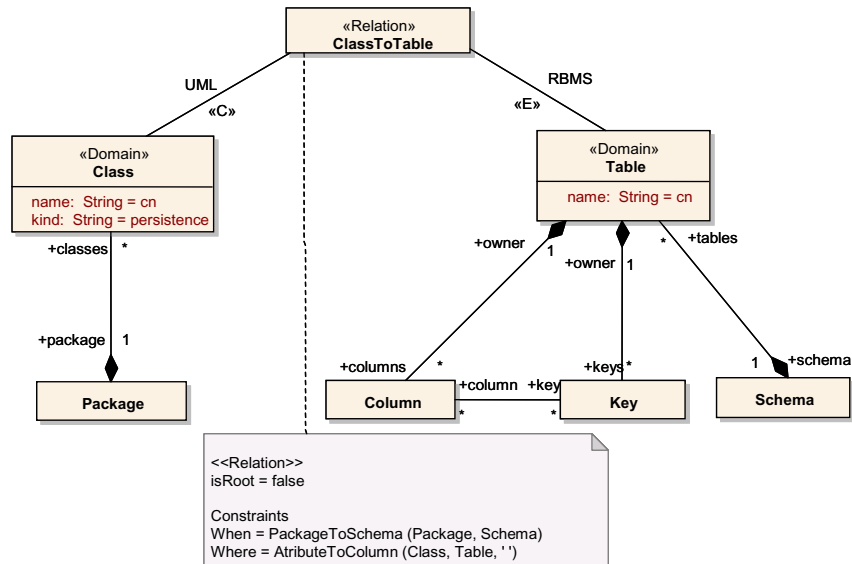
Fig. 5.   Example of the ClassToTable UPT Relation

The UPT Tool

The Model transformation tools are recognised as being of crucial importance for the Model Driven approaches to be carried out successfully. Our approach presents a Web tool developed on J2EE platform which implements the UPT model transformations. The UPT tool is based on the standards provided by the OMG (UML, XMI, MOF and OCL) mainly for two reasons: (1) optimize the implementation effort, through off-the-shelf components such as parsers, compilers, classes and frameworks provided by these standards, (2) facilitate the use of any type of UML tool that possesses the support for class diagrams.

The UPT Tool is a Web application and can therefore be used in a remote way. This tool does not need to provide a modelled graphic interface, because models and transformations are specified from any other UML tool and it is from them that their representation can be generated in XMI. The XMI is the input and output that the UPT provides.

Now, Fig. 6 describes in a generic form the application architecture, made up of different components, each of which performs a task within the process of transformations. We will start the description in a gradual form following the process by which UPT transformations are performed.

It consists of the following steps:

1. First, the MOF source metamodel and the MOF target metamodel are defined. This task is realized using a UML tool able to represent these metamodels and export them to XMI document for MOF metamodels (XMI-MOF).
2. Transform the metamodels defined in XMI document for MOF into JMI (Java Metamodel Interface) metamodel [5]. The use of JMI has two advantages: (1) it

manages the instances of the MOF metamodel classes, that is, it makes it possible to manage models which are compliant with the metamodels, and (2) it provides the parses for reading and writing from XMI documents for UML that represent the models. This is a semiautomatic step which is provided by frameworks like MDR of Netbeans [10].

3. We define the UML transformations following the UPT notation in any UML tool that may generate the XMI documents for UML.

4. The tool imports the XMI of the UPT transformations and from them it initiates the process of compilation in order to obtain the transformation in Java code and its subsequent compilation. Each transformation converts instances of the JMI origin metamodel to instances of the JMI target metamodel. Here ends the tool phase of metamodels definition and transformations. From this point we start to use the tool for the execution of transformations.
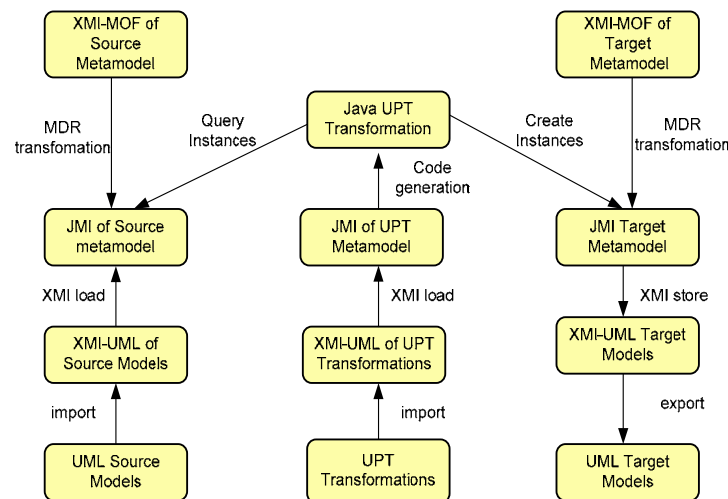

Fig. 6.   The Architecture of the UPT Tool

5. The process goes on to define the origin models in a UML tool from which the XMI for UML will be generated. This XMI will be imported by the parsers of JMI, and thus our metamodels in JMI will be charged with the models that have been introduced.

6. The UPT transformations are executed to transform the instances of JMI origin metamodel into the instances of the JMI target metamodel. Finally, the instances of the target metamodel are exported to XMI document for UML with the writers that JMI provides.

Whoever wants to use the UPT tool must be aware of the fact that there are two restrictions: (1) both origin and target metamodels of transformations must be defined in MOF. (2) The models of the approaches must also be defined in UML. This tool has been currently applied with success on the MDA approach called WebSA [9], which defines a development process for the complete specification of Web applications. This approach defines a PIM-to-PIM transformation that permits the

integration of the functional models provided by approaches such as OO-H [3] or UWE [8] and the architectural models of WebSA. This integration model allows us to obtain the final implementation.

Related Work

In addition, we will review the most relevant proposals of transformation languages that provide some implementation.

ATL [2] is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations. It is described by an abstract syntax (a MOF metamodel), a textual concrete syntax and an additional graphical notation allowing modellers to represent partial views of transformation models. ATL has been compared with QVT in [6] and the results show a reasonable compatibility between the two. However, QVT is the standard and there is a bigger community of experts in the OMG proposal. This approach presents an implementation in the Eclipse GMT (Generation Model Transformation) subproject [1].

UMLX [20] is another well-known approach. It presents a graphical transformation language that integrates with UML to define a mapping between schema instances supported by the GMT Tool [19]. UMLX presents an experimental concrete syntax for a transformation language. In contrast with UPT, UMLX does not follow the standard QVT. It is not defined as a UML profile and cannot be represented in any UML tool.

Finally, the MT [18] language is a unidirectional model transformation language, implemented as a DSL within Converge. MT defines an embedding of model transformations using declarative patterns to match against model elements whilst allowing imperative transformations. Unlike the UPT approach, MT cannot be expressed in a UML graphical tool because it must be represented in the Converge proprietary tool.

Conclusions and Future Work

In this paper, we have described a new proposal called UPT which improves the current graphical notation of QVT approach through a complete compatible UML notation. The UPT language defines a simplified metamodel which allows the definition of a UML profile of the class diagram. Thus, UPT offers a great tool compatibility because the transformations can be represented in any UML tool and can also be shared using XMI.

This work has been completed with the UPT tool which provides a Web interface mechanism to receive the models and transformations using as input the XMI of the UML models. Today, this tool has been applied to an MDA proposal as WebSA but it could be applied to any approach which defines its models in the standard UML.

In our future work, we will try to introduce in the UPT tool the transformations model-to-text in order to complete the tool and obtain the final implementation of the models. We also want to improve the process of converting the XMI-MOF

metamodel into JMI. Our aim is to make this conversion automatic. This improvement would allow any MOF-compliant approach to benefit from the UPT tool.

References

1. ATL GMT subproject. http://www.eclipse.org/gmt/atl/. 2006
2. Bézivin, J, Jouault, F, and Touzet, D : An Introduction to the ATLAS Model Management Architecture. Research Report LINA, (05-01).2005
3. Cachero C. OO-H. Una extensión de los métodos OO para el modelado y generación automática de interfaces hipermediales, http://www.dlsi.ua.es/~ccachero/pTesis.htm, 2003
4. Dobing B., Parsons J. Current Practice in the Use of UML. 1st Workshop on Best Practices on UML. ER Workshops 2005, LNCS 3770, October 2005
5. Java Metadata Interface (JMI). http://java.sun.com/products/jmi/. 2006
6. Jouault, F, and Kurtev, I. On the Architectural Alignment of ATL and QVT. In: Proceedings of ACM Symposium on Applied Computing (SAC 06), model transformation track, Dijon, Bourgogne, France, 2006
7. Kent S. Model Driven Engineering. IFM 2002, LNCS 2335, pp.286 –298, 2002
8. Koch N., Kraus A. The Expressive Power of UML-based Web Engineering, In Proc. of the 2nd. IWWOST, 105-119, CYTED, Málaga, Spain, June 2002
9. Meliá S., Gomez J., Koch N. Applying Transformations to Model Driven Development of Web applications. 1st International Workshop on Best Practices of UML (ER, 2005), LNCS 3770, October 2005
10. Metadata Repository Project HOME (MDR) http://mdr.netbeans.org/architecture.html. 2006
11. OMG MOF 2.0 Query/Views/Transformations. Request for Proposals. OMG doc. ad/02-04-10
12. OMG. Meta Object Facility (MOF) v1.4, OMG doc. formal/02-04-03
13. OMG. Model Driven Architecture, OMG doc. ormsc/2001-07-01
14. OMG. MOF Query/Views/Transformations Draft Adopted Specification: OMG doc. ptc/05-11-01
15. OMG. Object Constraint Model (OCL). OMG doc. ad/2003-01-07
16. OMG. UML 2.0 Superstructure Specification. OMG doc. formal/05-07-04
17. OMG. XML Metadata Interchange (XMI) Specification. OMG doc. formal/03-05-02
18. Tratt L. The MT Model Transformation Language. Proc. ACM Symposium on Applied Computing, pages 1296-1303, April 2006
19. UMLX Development Tools GMT subproject http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/UMLX/index.html. 2006
20. Willink E.D. UMLX: A graphical transformation language for MDA. In pp. 13-24, 2003

Pattern-to-Pattern Transformation in the SECTET

Muhammad Alam and Ruth Breu

Quality Engineering, University of Innsbruck
Austria
{muhammad.alam,ruth.breu}@uibk.ac.at
(Position Paper)

Abstract. In this paper we extend the transformation component of the SECTET-framework within the Authorization Module to incorporate a highly flexible and modular structure for the specification of complex constraint patterns. Based on the Query View Transformation (QVT) specification, the transformation component uses a very intuitive way to transform high-level complex constraint patterns to low-level XACML element patterns.

1 Introduction

Transformations represent an integral aspect of the Model Driven Architecture (MDA) paradigm and are used to generate target Platform Specific Models (PSM) from the source Platform Independent Models (PIM). The OMG has recently adopted a specification for the transformation between dissimilar meta-models called *Query View Transformation* (QVT). The SECTET-framework [4] – an inter-organizational workflow (Model-Driven) Security Engineering framework employs the QVT for the transformation of high-level security models to low-level security artefacts responsible for the configuration of underlying security architecture. The main innovative features we present in this paper are 1) the support for a highly flexible and modular structure to specify complex constraint patterns of a Domain Specific Language - SECTET-PL [5, 3] and 2) the inclusion of an additional layer of abstraction for the sake of facilitating transformations between the high-level SECTET-PL constraint patterns and the low-level XACML [6] element patterns. SECTET-PL is a predicative language in OCL-style for the realization of dynamic security requirements like access control [3], privacy-enhanced access control [1] and attribute-based delegation of rights [2]. SECTET-PL predicates perform the property check of the Document Model which describes the user profiles and documents travelling between the business partners in the form of a UML class diagram. These predicates are realized through various Access Requirement Models (ARM) such as *Access*, *Privacy* and *Rights Delegation Model*. SECTET-PL predicates are conditions under which the operations of the (web) service defined in the Interface Model as an abstract set of UML operations are accessible to roles defined in the Role Model as a UML class diagram. At the meta-level, the *Domain Model* (cf. Fig 1) defines the abstract syntax of SECTET-PL predicates using the Meta Object Facility (MOF). Further,
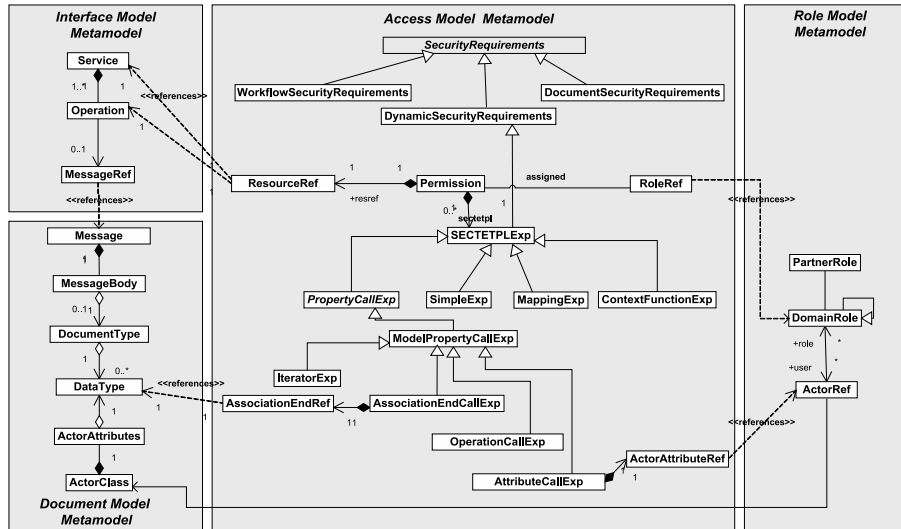
Fig. 1. Simplified Domain Model (some attributes are omitted for brevity)

it provides the integration between the business requirement models and the access requirement models with a focus on resolving model dependencies through proxy classes (such as `ActrorAttributeRef`, `AssociationEndRef` etc). At the instance level, each predicate is identified through some unique characteristic e.g. all predicates having a `NotEmpty` Boolean set operation are identified as `NotEmptyPattern`. Figure 2a shows an instance of the Domain Model for an
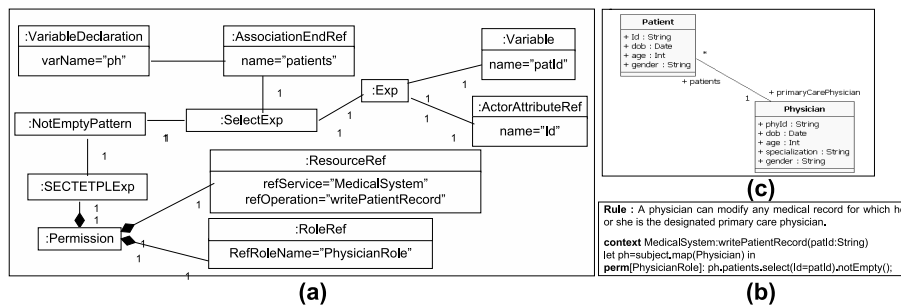
Fig. 2. a) Domain Model Instance **b)** Sample Access Model **c)** Sample Document Model

example from the medical domain. According to this example (cf. Fig 2b), "*a Physician is only allowed to modify a medical record if he/she is a primary care physician*". The `subject.map` function 1) abstracts the details of the authenti-

cation 2) assigns a role to the calling subject based on his/her credentials and 3) maps it to an internal representation in the *Document Model* (cf. Fig 2c – `Physician`). The SECTET-PL predicate - identified as `NotEmtpyPattern` at the domain-level navigates through the structure of the Document Model and performs the required model property check. In this way, the Domain Model provides a common syntactical and semantic base for the SECTET-PL expressions. Our
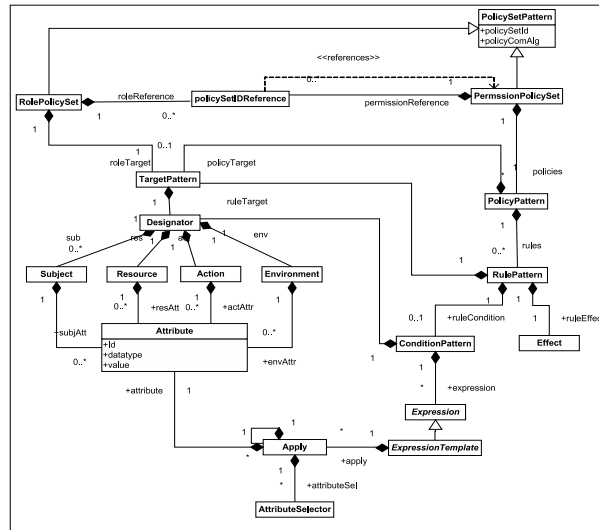

Fig. 3. XACML Meta-model (composed of XACML Element Patterns)

domain specific XACML policy meta-model (cf. Fig 3) at the PSM abstraction level is composed of various XACML element patterns e.g. `ConditionPattern`, `PolicyPattern` etc. The following section describes a pattern-based transformation of SECTET-PL predicates to XACML elements using the relational QVT transformations.

2 Pattern-to-Pattern (P2P) Transformations

Due to the semantic variations between the source (SECTET-PL) meta-model and the target (XACML) meta-model, we introduce an additional layer of abstraction between the source and the target models. This means that, in order to make SECTET-PL patterns compliant with the underlying security architecture PSM, SECTET-PL patterns are first transformed to transformable SECTET-PL patterns. This approach has two advantages. First, one security PIM can be transformed to multiple security architecture PSM at the same time. Secondly, the length of the QVT scripts is greatly reduced due to the transformation of high-level SECTET-PL patterns to transformable SECTET-PL patterns
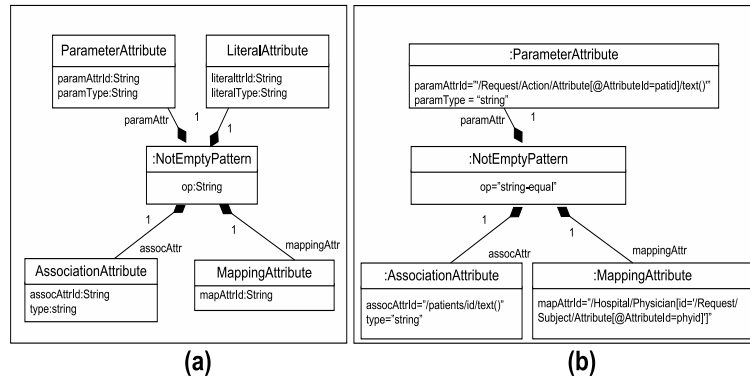
Fig. 4. a) Transformable `NotEmptyPattern` and **b)** its Instance

(cf. Fig 4a). Our prototypical tool [5, 3] uses the ANTLR citeANTLR, a compiler program to generate abstract syntax tree (predicate patterns) from the SECTET-PL predicates. The tool performs the semantic analysis of the predicate patterns against the model information specified via XMI files. The semantic analysis phase is also used to populate the transformable patterns with the SECTET-PL pattern instance (cf. Fig 2a) as well as domain specific values (cf. 4b). Figure 5 shows an example *Transformation Pattern* that uses

Fig. 5. Sample Transformation Pattern

the *Relations* language of the QVT. The pattern transforms the Transformable SECTET-PL (tsectetPL) patterns to XACML element patterns. The transformation `tsecetetPLToXACML` (line 1) defines two typed candidate models (as parameters): "`ts`" of type `tsectetPL` and "xacml" of type `XACML`. In order to have successful transformation from `ts` to `xacml`, the set of relations defined within this transformation must hold. For example, the transformation `tsectetPLToXACML`

contains a relation `NotEmptyPatternToXACMLCondition` (line 3) which defines two domains `NotEmptyPattern` from the domain `ts` (line 6) and `ConditionPatte rn` from the domain `xacml` (line 20). According to QVT, these domains define distinguished typed variables that are used to match a model of a given model type and are called patterns. For example, the domain `NotEmptyPattern` (line 6) contains a pattern which matches with the model element `NotEmptyPattern` within the domain `ts`. The `op` attribute in `NotEmptyPattern` is bound to the variable `operator` (line 7) and `assocAttrId` attribute of the `AssociationAttribute` is bound to the variable `subjAtrrId` (line 9). These variables are used to exchange information between meta-models e.g. the variable operator is used in the domain `ConditionPattern` (line 21) to make a pattern of the form `functionId=operator`. This pattern implies that the relation `NotEmptyPatternToXACMLCondition` will only hold if `NotEmptyPattern` has the same `op` as `operator` and `ConditionPatte rn` has the same `functionId` as `operator`. These domains can contain nested patterns as well e.g. `AssociationAttribute` defines a nested pattern which bounds the value of the `assocAttrId` and `assocAttrType` attribute to the `subjAttrId` and `subAttrType` variables respectively.

3   Conclusion

In this paper, we presented a novel approach for the transformation of complex constraint patterns of a Domain Specific Language - SECTET-PL to web services security standard artefact XACML. The approach can be applied to any web service security standard. The high-level SECTET-PL patterns are transformed to low-level XACML element patterns in a very intuitive way. The modular Pattern-to-Pattern transformations give great benefits ranging from strong traceability between high-level and low-level security artefacts to the involvement of all stakeholders - from the domain expert to the software engineers.We are currently extending our tool support to incorporate the QVT-based, visual Pattern-to-Pattern transformations.

References

1. M. Alam, M. Hafner, and R. Breu. Modeling Authorization in a SOA based Application Scenario. IASTED Software Engineering 2006, ISBN: 0-88986-572-8.
2. M. Alam et al. MA Framework for Modeling Restricted Delegation in Service Oriented Architecture. To Appear in Trust Bus 2006.
3. M. Alam et al. Modeling Permissions in a (U/X)ML World. In IEEE ARES 2006, ISBN: 0-7695-2567-9.
4. M. Hafner et al. "SECTET An Extensible Framework for the Realization of Secure Inter-Organizational Workflows". Accepted for ICEIS 2006.
5. SECTETPL : A Predicative Language for the Specification of Access Rights. http://qe-informatik.uibk.ac.at/~muhammad/TechnicalReportSECTE TPL.pdf.
6. XACML 2.0 Specification Set. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

Automating Metamodel Mapping Using Machine Learning

Jamal Abd-Ali, Karim El Guemhioui

Department of Computer Science and Engineering
University of Quebec in Outaouais (UQO)
C.P. 1250 succ. Hull, Gatineau (Quebec), J8X 3X7 Canada
{abdj01,karim}@uqo.ca

Abstract. Model transformations are at the heart of the MDE approach. In this paper, we make the case that provided the availability of corresponding models expressed in different formalisms, we can build on some AI results related to natural language translation to learn about and automate model transformations.

1 Introduction

Models are means of describing real systems we intent to study, and of specifying engineering systems we plan to build. For every domain, we can define specific modelling elements that represent the main concepts of that domain. The modelling elements supply the vocabulary used in the domain, while the domain concepts hold the semantics of the modelling elements. The vocabulary augmented with structuring rules and other constrains form a specific modelling language, also called a metamodel. More accurately, models are described using a modelling language whose model is the metamodel. Several metamodels have already been published: the Unified Modelling Language [9], the Enterprise JavaBean metamodel [10], a .NET component metamodel [1], etc. Note that individually each model element is an instance of its corresponding metamodel element; whereas, collectively the model elements must conform to the structure and requirements of their metamodel.

In the artificial intelligence (AI) field, several learning methods have been devised that have permitted advances in the automatic translation of natural languages [3], [6]. The basic idea we want to explore is inspired by these AI approaches and aims to use machine learning techniques [4] to discover transformation rules between existing models of the same system, and then infer automatic mappings between their metamodels. There is indeed a strong analogy between, on the one hand, a natural language and its grammar, and on the other hand, a model and its defining metamodel. For example, in an English text (the expression language), the words and their arrangement must comply with the rules of the English grammar. English can be considered as the metamodel and English texts as models. But contrarily to natural languages, modelling languages are more contrived, and usually finite, unambiguous, well formed, etc. Therefore, our approach shouldn't suffer from the limitations AI has encountered with natural language translation.

2. Hypotheses

Let MM1 and MM2 be two languages for expressing models (i.e. metamodels). We assume MM1 and MM2 are each made of a finite number of elements. We also assume that these two metamodels enable the expression of the same semantics, though each one may have its own elements structured according to different rules.

To describe the same system, we can produce two models M1 and M2, expressed respectively in MM1 and MM2. We say that M2 is a translation of M1, and conversely. We express that fact by a correspondence relationship between M1 and M2 that maps specific elements of M1 to their correspondents in M2. The function that converts M1 into M2 is also called a translation transformation or translation function since it preserves the meaning of M1 and "transfers" it to M2.

Note that while describing the same system, each one of the two models M1 and M2 represents this same reality with its own limitations due to the (lack of) expressiveness of the respective languages (the MM1 and MM2 metamodels). Therefore, we prefer to say that M1 and M2 are correspondent rather than equivalent; we cannot guaranty the total faithfulness of the translation nor its reversibility.

Note that the need to generate corresponding models (model translations) is a recurring and challenging problem in several areas of software engineering; mainly in model driven development [2], [8], [11], or when faced with model rewriting for integrating or merging models that describe parts of a single system or several related systems.

Following in the AI spirit, we assume that we dispose of a knowledge base of translated models (the corresponding models). More specifically, we assume that we are given a number m of couples of corresponding models represented by the set S = {(Mi1, Mi2) such as $1 \leq i \leq m$}. We are interested in defining a function f that takes as input an M1 model expressed in MM1, and produces as output an M2 model representing the translation of M1 with respect to the MM2 metamodel.

3. The Approach

Starting from the available sample S, we need to discover the underlying model T embodying the logic behind the correspondences in the data sample. Building or simulating this model will lead us to explore and, if satisfied with the compliance, add new data to the initial sample S. We propose to resort to a Machine Learning approach. Such approach relies on the understanding of some phenomenon and on a sample of data and hypotheses about this phenomenon, to build a model that enables the simulation of the phenomenon under study.

In our case, the sample consists of a set of couples (a_i, b_i), where a_i represents a model Mi1 written in terms of MM1, and b_i represents the corresponding model Mi2 written in terms of MM2. In the context of this work, the model T is in fact a function f that receives a_i as input and produces the corresponding b_i as output. Since there is no guaranty that the sought after function f can be found, the machine learning process will consist in searching for an estimate of the function f that recoups the sample S, and that relies on an inductive reasoning to produce the right output for

every input not in the sample. Such learning is said cognitive and its convergence towards the sought after function is the principle behind the Machine Learning approach which uses several techniques grounded on mathematics and statistics.

With respect to our sample S, we want to estimate the function f which is a transformation converting models expressed in MM1 into corresponding models expressed in MM2. According to our approach of Machine Learning, we need to guide the search by proposing a space H of hypotheses about the function f, derived from some of its known characteristics (e.g., we know that the function generates translations at the model level, so it defines mappings at the metamodel level) . The search of f will be limited to the H space. We need also to make some hypotheses about the language used to express f, which we will call LH. An example of hypothesis about LH could be related to the formalism we intend to use to represent f (first order logic, a numerical function, etc.)

LH and H are hypotheses we make based on our knowledge and understanding of the application domain of f and on the information we may withdraw from the sample S. For example, if we know that the function we are looking for is a numerical function in a polynomial form of degree 5, this defines H and LH and confines the search to using the sample to solve 6 equations linearly independent allowing the discovery of the sought after function with a convergence usually guaranteed with classical numerical methods. This illustrates that the more we know about H and LH the simpler and more guarantied is the finding of f.

With respect to our sample S, we can for example assume that the function f is defined by a succession of rules of corresponding patterns between MM1 and MM2. We mean by pattern an arrangement of metamodel elements conform to their metamodel, in addition to other constraints implicating these elements and characterizing the pattern.

We can make this assumption based on our knowledge of the model driven development domain and of the work on metamodeling in general, where we can find many defined model transformation functions [2], [5] that enable us to suggest effective hypotheses about the sought after function f. This assumption is also inspired by natural language translation techniques that relate a word in one language to a corresponding word in another language, or a sentence to a corresponding sentence, etc.

The estimation of f relies on a well defined expression language LH and a space H; and the search for f benefits from the knowledge we gain from the S sample which implies that $f(Mi1) = Mi2$ for each i between 1 and m. The f function is composed of a finite set of declarations of corresponding patterns between MM1 and MM2. Note that the number of elements of a pattern is comprised between 1 and the total number of elements of the enclosing metamodel. A learning algorithm can start by searching pattern correspondences made of a single element, then made of two elements, and so on, until we reach a maximum, which is the number of elements of the metamodel with the highest number of elements.

3. Discussion

A lot of work still needs to be done regarding the definition of the best space H of hypotheses about f, as well as the LH language for expressing f in a formalism suited to a learning algorithm.

### 3.1 Towards a Pattern Correspondence Formalism

The expression of a rule of correspondence between two patterns can be reduced to a vector V whose elements are Boolean. The dimension of V is equal to the total number of elements (in a large sense, including associations) in the MM1 and MM2 metamodels. Let a_i (with i=1,..,k) and b_j (with j=0,…,h) be the elements of the two metamodels, respectively MM1 and MM2. We will have:
$V = [\alpha_1, \alpha_2, …, \alpha_k, \beta_1, ..., \beta_h]$; where the elements a_i with $\alpha_i=1$ and b_j with $\beta_j=1$ denote corresponding elements in the two patterns expressed in the vector V (conversely, $\alpha_i=0$ means that a_i is not part of the pattern correspondence).

If we consider a space H of hypotheses about f that limits to p the maximum number of corresponding patterns, we can represent f as a matrix **M** with p rows. The p rows of the matrix are vectors (like V) of dimension k+h. Actually, this is only a simplistic attempt to provide a specific representation of corresponding patterns, because we haven't taken into account several pattern specifications like:

- Contextual constraints that can put some restraints on elements of the source and target models as a necessary condition to trigger a pattern correspondence rule;
- The application of filtering rules to select certain instances of patterns and ignore others when applying a correspondence rule;
- The possibility of having several instances of one metamodel element in the same pattern. Obviously, the vector V will no longer be enough to represent such a pattern rule, because it doesn't provide for the repetition of the α's or β's denoting the instances of the same metamodel element. Furthermore, an association cannot be fully specified by only indicating its type in the metamodel; both association ends must be specified.

Relying on what we have said and as the next step in the definition of H and LH, we could proceed by establishing a protocol and a searching algorithm for the estimation of the function f. The figure that follows gives an architectural overview of the approach.
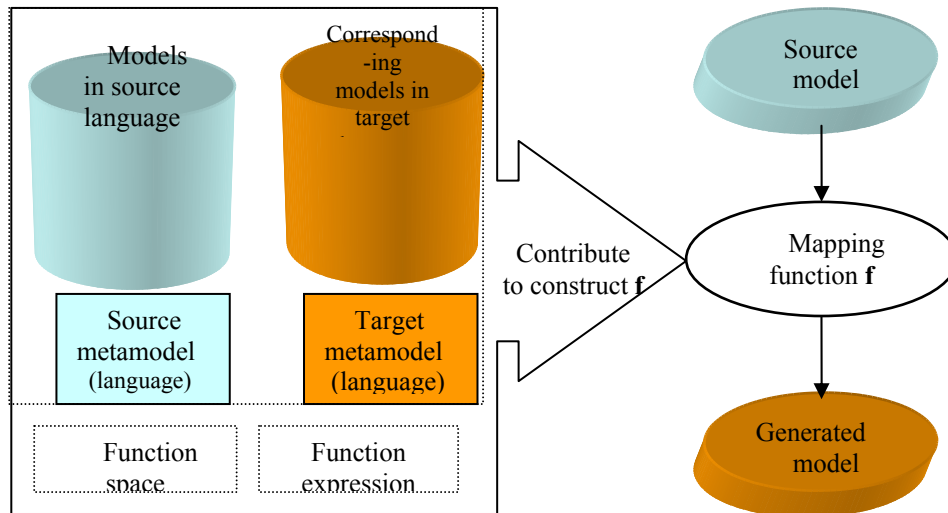
3.2 Convergence and Success

The success of the approach depends on two main factors: (1) the choice of the space of hypotheses (H) using a suitable learning technique, and (2) the knowledge contained in the sample S.

As a technique that suits the mentioned formalism, we propose the Candidate Elimination Algorithm. The idea is to consider all the possible values for the elements M_{ij} of the matrix M and confront the resulting vectors V with the sample S.

By considering all the possible values M_{ij}, we mean that we take into account all the possible patterns in both metamodels, and we try to explore mappings between them relying on available examples in the sample S. We will probably have a huge number of candidate vectors CV as potential vectors V of the matrix M. To avoid a combinatorial explosion in the number of candidate vectors CV, we can limit our search to patterns composed of a limited number of elements, and limit the number of rules of corresponding patterns.

To assess the probability that a CV is a good candidate, we can append to each CV a couple of elements (g, b) that indicate the number of compliant and non compliant examples with the considered CV. At the end, we can obtain the best CV by comparing the values g and b of all the candidate vectors. Using the results we obtain when we limit the pattern size and the number of rules, we can assess the situation with respect to the completeness of the learned transformation. We can then decide to go for patterns with more elements and explore a more limited number of mapping rules. We can also attempt an interpretation of a retained CV with non zero b's, by finding the conditions that enable or disable a rule. Theses filtering or enabling conditions are not covered in the matrix M.

In any learning process, the system that is learning withdraws the added knowledge from the data sample at its disposal. The size of the data sample is significant for deducting the model that governs these data. For example, a single data illustrating a correspondence rule is not enough to identify and decipher the rule. Even if we could detect the rule, an inductive reasoning cannot rely on a single verification of the rule hypothesis to generalize it.

Furthermore, even if the sample size is fairly large, it should not represent a redundancy of some partial aspect of the model we are looking for (in our case: a function f). An extreme situation would be that all the data of the sample S could be deducted from one of its records, as if all the data illustrate the same pattern correspondence rule; the remaining rules of f will then be undetectable.

A completely different approach would be to determine the transformation rules based on a comparison of the semantics of the elements of both metamodels MM1 and MM2. Validation of such an approach raises very challenging problems [7] since it must take into account the semantics of both metamodels and the definition of the transformation, which are usually written with different formalisms and often with semantics in informal text attached to metamodel elements.

Our goal is to preserve the semantic content of the source models according to a learning technique that evolves using the results of many guessing approaches provided as a sample of corresponding models.

According to a machine learning approach, the discovery of rules relies on scientific knowledge in mathematics and statistics, which enables the assessment of the quality of the obtained results. Thus, the convergence towards an optimal function that best represents the reality depends mainly on the following factors.

- The size of the data sample and the extent to which it covers the knowledge we are seeking. This coverage will allow the detection of all the rules constituting the definition of the function f we are looking for.
- The assumptions made about the expression language of the function, and the space in which we need to search for the function.

If these factors are satisfied, we expect, according to the principle behind machine learning that the search of the f function will boil down to choosing the appropriate algorithm that converges towards the right function. The quality of the f function can be assessed by testing to which extent f complies with the sample S used to find it, and to which extent f complies with another test sample different from S. Furthermore, the definition of convergence itself depends on the definition of a distance between the outputs of f (which is a distance between models). Defining such a distance is an essential part of the research work that this paper aims to lay out.

Conclusion

Our approach can take advantage of several studies regarding the data of samples under consideration. We believe that it confers to them some complementary. We should also benefit from a capacity of continuous evolution related to new samples, as is usually the case in AI approaches.

We can use our approach to assess or even benefit from an existing model transformation definition, by adopting it as a starting point in the machine learning

process. We can then study the impact of this starting definition on the convergence of the process. We can also study the modifications brought to the starting point function by the learning process.

We expect a significant amount of work in the development of a framework for specifying the hypotheses space of the models' transformations, the formalism the most suited to their expression, and some metrics such as the distance between models. The assessment of the approach awaits also some experimental work using samples of significant sizes for machine learning.

We believe that our approach will be of interest to researchers working in the area of model engineering, application integration, and even to developers of automatic translators of formal or natural languages.

References

1. Abd-Ali, J., El Guemhioui, K. "An MDA-Oriented .NET Metamodel". 9th IEEE International Enterprise Distributed Object Computing Conference (EDOC'2005), Enschede, The Netherlands, 2005. pp. 142-153

2. Abd-Ali J., El Guemhioui, K. "Horizontal Transformation of PSMs". European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA), Nuremberg, Germany. 2005. LNCS 3748 pp. 299-315.

3. Chichester, Ellis Horwood. Artificial Intelligence Systems, in Machine Translation: past, present, future (Ellis Horwood Series in Computers and their Applications.), 1986. 382p.

4. Cornuéjols, A., Miclet, L., Kodratoff, Y. Apprentissage artificiel, concepts et algorithmes. Eyrolles 2002. 591p.

5. Fleurey, F., Steel, J., Baudry, B. "Validation in Model-Driven Engineering: Testing Model Transformations". Workshop WS5 at the 7th International Conference on the UML, Lisbon, Portugal. 2004. Available @ http://www.metamodel.com/wisme-2004/papers.html

6. Hutchins, J. Machine translation, (computer-based translation), Publications by John Hutchins available @ http://ourworld.compuserve.com/homepages/WJHutchins/

7. Kleppe, A., Warmer, J. "Do MDA Transformations Preserve Meaning ? An investigation into preserving semantics". First international workshop on Metamodeling for MDA, York, UK, 2003.

8. OMG: MDA Guide Version 1.0.1 document number omg/2003-06-01 available at http://www.omg.org/docs/omg/03-06-01.pdf

9. OMG UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel. http://doc.omg.org/ptc/2004-10-05

10. OMG. Metamodel and UML Profile for Java and EJB Specification. February 2004. Version 1.0, formal/04-02-02. An Adopted Specification of the Object Management Group, Inc.

11. Model Driven Engineering (MDE). http://idm.imag.fr/