UNIVERSITY OF BELGRADE
FACULTY OF ORGANIZATIONAL SCIENCES

MILAN V. MILANOVIĆ

MODELING RULES ON THE SEMANTIC WEB

MASTER THESIS

Belgrade, 2007.

Modeliranje pravila Semantičkog Web-a

Apstrakt:

U okviru ove magistarske teze je predstavljen dizajn i razvoj generalnog jezika za razmenu pravila, što treba da omogući interoperabilnost različitih vrsta pravila na Semantičkom Web-u, kao i razvoj i implementacija integracije više jezika za predstavljanje pravila na Semantičkom Web-u i Model Driven Arhitekturi (MDA). Postojeća rešenja integracije jezika za predstavljanje pravila ne koriste jedan centralni jezik za predstavljanje pravila, što značajno utiče na mogućnost jedinstvenog predstavljanja i korišćenja pravila. Ovo onemogućava razmenu pravila u različitim softverskim sistema koji ih koriste. Pored toga to utiče i na broj transformacija koje je neophodno koristiti i razviti. U tim rešenjima se vrši integracija ovih jezika transformacijama između njihovih konkretnih sintaksi pomoću XML baziranih alata. Posledica je da ovo onemogućava fokusiranje na mapiranjima između različitih jezičkih konstrukcija, već je implementacija orijentisana na implementacione detalje njihovih konkretnih sintaksi. Pored toga, aktuelna rešenja ne nude mogućnost integracije jezika zanovanih na XML-u sa predstavljanje pravila sa drugim jezicima koji nisu definisani XML sintaksom, kao što je npr. OCL.

Za integraciju jezika za predstavljanje pravila je razvijen REWERSE I1 Rule Markup Language (R2ML) kao centralni (generalni) jezik za razmenu pravila. R2ML omogućava predstavljanje različitih vrsta pravila, tj., ograničenja integriteta, izvedena, reakciona i produkciona. Implementacija integracije pravila preko R2ML-a je realizovana za W3C-ov Semantic Web Rule Language (SWRL) sa Ontology Web Language-om (OWL) i OMG-ov Object Constraint Language (OCL) sa Unified Modeling Language-om (UML). SWRL je jezik za predstavljanje pravila na Semantičkom Web-u, dok je OCL jezik za definisanje pravila nad UML modelima.

Integracija navedenih jezika korišćenjem R2ML-a kao “mosta” je implementirana transformacijama metamodela ovih jezika, gde svaki jezik za predstavljanje pravila ima apstraktnu sintaksu (koja se definisanu meta-modelom u kontekstu MDA). Kako se meta-modeli jezika za pravila nalaze u MOF tehničkom prostoru konkretne transformacije su zasnovane na MOF2 QVT standardu OMG grupacije. Korišćen je ATLAS Transformation Language (ATL) koji predstavlja implementaciju MOF2 QVT standarda za transformaciju između modela.

U ovom radu su prikazana konceptualna mapiranja između elemenata SWRL, R2ML i OCL jezika. Kako se konkretne sintakse jezika nalaze u različitim tehničkim prostorima (npr., XML, MOF, EBNF) kreirane su neophodne transformacije korišćenjem ATL jezika za prelaz između ovih različitih tehničkih prostora i MOF-a kao centralnog tehničkog prostora u kojem se transformacije izvršavaju i u kojem su definisane njihove apstraktne sintakse (tj. meta-modeli).

Projektovana je i razvijena programska biblioteka u jeziku Java za izvršavanje transformacija u standardnim i Web aplikacijama. Korišćenjem ove biblioteke u Java Server Pages (JSP) jeziku razvijena je Web aplikacija za izvršavanje transformacija na konkretnim pravilima reprezentovanim različitim jezicima za predstavljanje pravila. Pored opisa dizajna i implementacije razvijenog softverskog rešenja, u radu je dat koncizan pregled istraživačkih disciplina relevantnih za realizovano istraživanje. Takođe, prikazane su prednosti pristupa za transformacije korišćenja MOF2 QVT standarda u odnosu na druga rešenja.

Ključne reči:

Semantički Web, Ontologije, Jezici za predstavljanje pravila, Meta-modeli, Transformacije između meta-modela, Tehnički prostori, SWRL, OWL, UML, OCL, XML, EBNF, R2ML, MOF2 QVT, ATL.

Modeling Rules on the Semantic Web

Abstract:

This master thesis focuses on the design and development of a general markup rule language for sharing rules. This language should enable interoperability between different types of rules on the Semantic Web and those defined by standards (e.g., Meta-Object Facility, MOF) of the Model Driven Architecture (MDA). Currently available solutions for integration of such languages do not use a disciplined solution based on a central rule markup language, which might have a significant influence on potentials for unique representation and reuse of rules. Besides, this affects a number of transformations necessary to develop and use. Morover, integration between rule languages is usally done on the level of their concrete syntax with the use of XML-based tools. This prevents focusing on mappings between constructs of rule languages at the level of their abstract syntax, but, instead one should focus on the implementation level, that is, the level of concrete syntax. In addition, the current solutions are hampered with the lack of tools for integration between XML-based rule languages with other languages that do not have an XML syntax such as the OCL language.

In this paper, we use the REWERSE I1 Rule Markup Language (R2ML) to allow for integration of rule languages, where R2ML plays a role of a central (general) markup language for representing rules. R2ML can represent different types of rules, namely, integrity rules, derivation rules, reaction rules, and production rules. The use of R2ML is demonstrated on the integration of W3C's Semantic Web Rule Language (SWRL) that is based on the Ontology Web Language (OWL) and OMG's Object Constrant Language (OCL) that is based on the Unified Modeling Language (UML). SWRL is a language for representing rules on the Semantic Web, while the OCL is a language for defining rules in UML models.

Transformations between R2ML and other rule languages are implemented by employing the transformations of MOF-based models of these languages. In this way, the mappings are done at the level of the languges’ abstract syntax, as meta-models are actually abstract syntax. Since meta-models are located in the MOF technical space, transformations are based on OMG's MOF2 QVT specification. The implementation is done by using the ATLAS Transformation Language (ATL), an implementation of the MOF2 QVT specification.

This thesis shows conceptual mappings between different elements of the SWRL, R2ML and OCL languages. As the concrete syntax of these rule languages is located in different technical spaces (i.e., XML, MOF, EBNF), necessary transformations are created, as well as concrete textual syntax (for OCL) for bridging between these tehnical spaces and MOF, as a central technical space in which transformatons are executed.

In addition, a Java software library for execution of the ATL transformations in regular and Web aplications is developed. This library is empolyed for the implemenation of a set of JSP pages for execution of rule transformations. Along with description of design and implementation of the developed software solution, the thesis provides a concise overview of all research disciplines relevant for the accomplished research. Furthermore, the thesis gives compares the proposed transformation approach based on the MOF2 QVT specificion to other relevant solutions.

Keywords:

Semantic Web, Ontologies, Rule Markup Languages, Meta-models, Model to model transformations, technological spaces, SWRL, OWL, UML, OCL, XML, EBNF, R2ML, MOF2, QVT, ATL

Acknowledgements

I would like to express my deep thankfulness to all of the people who have supported me during my research. In particular, I owe my sincerest gratitude to my supervisor, Prof. Vladan Devedžić who introduced me to research, gave me the opportunity to pursue my research ideas and provided me with many helpful discussions in the stimulating atmosphere of the GOOD OLD AI laboratory. I am very grateful to Prof. Dragan Gašević from Athabasca University in Canada, who guided my work from the very first day. He gave me the opportunity to conduct this research and helped me made the right strategic decisions at many forks along the way. He kept me on the track while allowing me to broaden my research horizon in tangential areas and helped me incrementally build up my understanding of the research area. I also want to thank him for reading this thesis and providing helpful suggestions.

I deeply thank Dr. Adrian Giurca from Brandenburg University of Technology at Cottbus in Germany for his great collaboration, support and many useful suggestions. His remarks were invaluable in extending and improving many results. I am also very grateful to Prof. Gerd Wagner, also from BTU at Cottbus, for keeping me on the right track with his advices and for traveling a great distance to attend my defense.

I would also like to thank Prof. Dragan Milićev and Prof. Siniša Vlajić from the University of Belgrade, for being on my examining committee.

I also deeply appreciate the great help of my colleagues from the GOAI laboratory, including, Jelena Jovanović, Marko Ribarić, Uroš Krčadinac, Vitomir Kovanović and Milan Stanković who have contributed to this thesis by reading it and providing helpful suggestions in the translation of thesis into the English language. My gratitude is also directed towards Dr. Frédéric Jouault and Mariano Belaunde for helpful discussions on all aspects of model transformations and OCL.

A special mention deserves Dragan Tomaš, for supporting me in just as many ways. Thanks a lot!

Last but not the least; I would like to thank my parents Olga and Vojo who have always encouraged my work and believed in me, which has helped me get to where I am now. They supported me in everything I did and as much as they could throughout all my (long) student years.

CONTENTS

1. INTRODUCTION

The Semantic Web is based on the use of ontologies that should provide an explicit definition of the domain conceptualization [Berners-Lee et al., 2001]. Employing the rich AI research experience and being driven by practical needs for the use on the Web, the World Wide Web Consortium (W3C) has adopted the Web Ontology Language (OWL) as a standard ontology language [Dean & Schreiber, 2004]. Although the adoption of OWL means that Semantic Web applications can exchange their ontologies and tool vendors can develop reasoners and query languages over OWL, there is also a need to have some other mechanisms for defining knowledge. This is mainly manifested through advanced mechanisms for enriching ontologies by using rules. Thus, we should also define a standardized Semantic Web rule language that will be based on OWL to provide an additional reasoning layer on top of OWL. On the other hand, there are many Semantic Web applications that might use (OWL) ontologies whose business logic is implemented by using various rule languages (e.g., F-Logic, Jess, and Prolog) [Sheth, 2006]. In this case, the primary goal is to have a rule exchange language for sharing rules, and hence enabling reusability of their business logics.

The above arguments motivated the research in the (Semantic) Web community to look at their different aspects. The most important proposal for the first group of rule language is Semantic Web Rule Language (SWRL) [Horrocks et al., 2004] that tends to be a standardized reasoning layer built on top of OWL. However, this is just one submission to such a language, while the research on Semantic Web services (e.g., WSMO and SWSL) introduces/relies on other rule languages besides SWRL such as SWSL-Rules or F-Logic [Sheth, 2006]. In fact, this language diversity can be addressed by the second group of research efforts for Semantic Web rules manifested in the Rule Interchange Format (RIF) initiative [Ginsberg, 2006], which tries to define a standard for sharing rules. That is, RIF should be expressive enough, so that it can represent concepts of various rule languages. Besides RIF, one should also develop a (two-way) transformation between RIF and any rule language that should be shared by using RIF. Currently, there is no official submission to RIF, but RuleML [Hirtle et al., 2006] and the REWERSE Rule Markup Language (R2ML) [Wagner et al., 2006] are two well-known RIF proposals.

On the other hand, models are central concepts of Model Driven Architecture (MDA) [Miller & Mukerji, 2003]. Having defined a model as a set of statements about the system under study, software developers can create software systems that are verified with respect to their models. Such created software artifacts can easily be reused and retargeted to different platforms (e.g., J2EE or .NET). UML is the most famous modeling language from the pile of MDA standards [UML, 2004] [UML, 2005], which is defined by a metamodel specified by using Meta-Object Facility (MOF) [MOF, 2005]. MOF is a metamodeling language for specifying metamodels, i.e., models of modeling languages. Considering that MDA models and Semantic Web ontologies have different purposes, the researchers identified that they have a lot in common such as similar language constructs (e.g., classes, relations, and properties), very often represent the same/similar domain, and use similar development methodologies [Happel & Seedorf, 2006]. The bottom line is the OMG's Ontology Definition Metamodel (ODM) specification [ODM, 2006] that defines an OWL-based metamodel (i.e., ODM) by using MOF, an ontology UML profile, and a set of transformations between ODM and languages such as UML, OWL, ER model, topic maps, and common logics [ODM06]. In this way, one can reuse the present UML models when building ontologies.

In this thesis, we present a research on approaching the Semantic Web and MDA by proposing a solution to interchanging rules between two technologies. This integration will reduce human work and enable the reusing of well defined techniques. More specifically, this thesis addresses the problem of mapping between the Object Constraint Language (OCL), a language for defining constrains and rules on UML and MOF models and metamodels, and the Semantic Web Rule Language (SWRL), a language complementing the OWL language with features for defining rules. In fact, this proposal covers the mapping between OCL along with UML (i.e., UML/OCL) and SWRL along with OWL

(OWL/SWRL). The main idea of the solution is to employ the REWERSE Rule Markup Language (R2ML) [Wagner et al., 2006] (a MOF-defined general rule language capturing integrity, derivation, production, and reaction rules), as a pivotal metamodel for interchanging between OWL/SWRL and UML/OCL. R2ML is completely built by using model-driven principles (i.e., defined by a meta-model). This means that solution is completely based on the abstract syntax of both languages, unlike other similar approaches that mainly focus on a concrete syntax (mainly XML-Schema-based) without efficient mechanisms to check whether the implemented transformations are valid w.r.t. the abstract syntax. Presented solution also covers mappings between the abstract and concrete syntax of SWRL/OWL, UML/OCL and R2ML languages, which is completely decoupled from the transformations between the abstract syntax of these languages.

Since various abstract and concrete syntax are used for representing of these three metamodels (e.g., R2ML XMI, R2ML XML, SWRL (OWL) XML, OCL XMI, UML XMI, OCL text-based syntax), the implementation is based on the OMG's MOF2 QVT specification (which defines languages for model transformations) [QVT, 2005]. More specifically, we used Atlas Transformation Language (ATL) [ATL, 2006], a QVT implementation and one of official Eclipse model-to-model transformation engines[1], and by applying the *metamodel-driven model transformation principle*[2] [Bézivin, 2001].

Advantage of using the R2ML as the central meta-model, is that for N rule languages, among which we want to share rules, we need to create two transformations from a specific rule language to R2ML and from R2ML to the specific language, and thus the overall number of transformations to be implemented is 2N. If we create transformations for each pair of languages under study, then we need N*(N-1) transformations. Another advantage of the suggested solution is that it is possible to map rules from UML/OCL to other rule languages (e.g., Jess, F-Logic, Jena, RuleML and Prolog), for which are defined mapping with the R2ML, but not just to SWRL as it will be shown in this thesis.

1.1. RESEARCH GOALS

Based on the above mentioned facts, it is possible to formulate basic research goals that are presented in this thesis. Primary goals of this thesis are to show the actual achievements in the area of the Semantic Web and the Model-Driven Engineering, as well as to identify potentials for their integration and explore concrete technologies that can enable such an integration. This thesis shows potentials of using transformations between models of different concrete rule languages (UML/OCL, SWRL/OWL, R2ML) as one aspect of this integration.

This thesis also shows how is possible to enable integration between different metamodels of rule languages (i.e., OCL and SWRL), by using a general rule interchange language (i.e., R2ML) in the context of the MDA technologies. This also explores advantages and shortcomings of sharing different types of rules when using a central rule markup language (i.e., R2ML).

1.2. SUMMARY OF REALISED SOLUTION

In this thesis, we present a design and implementation of the rule interchange between different rule languages, by using the R2ML language as central "bridge". We participated in the development of the R2ML language, that is, its abstract syntax (i.e., meta-model) and concrete syntax. We also defined conceptual mappings between OWL/SWRL and R2ML and between R2ML and UML/OCL. These conceptual mappings are implemented by using QVT (or more precisely ATL) based transformations on the level of the abstract syntax (i.e., meta-model) of these languages. This approach allowed us to focus on the concepts of the languages considered, instead of on platform specific details that are usually integrated into languages' concrete syntax. Along with the integration of the languages mentioned, we also show how R2ML enables us to reuse previously implemented transformations

[1] http://www.eclipse.org/m2m/
[2] In this thesis we use the folowing abbreviation for this principle: *meta-model transformations.*

between R2ML and other languages such as and F-Logic, Jess and RuleML, and hence enable a further interchange of rules encoded in OWL/SWRL and UML/OCL.

In order to support sharing rules between OCL and SWRL via R2ML, we have to implement e transformations in different technological spaces (e.g., XML, EBNF, and MOF), because the concrete and abstract syntax of these languages is defined in all these technological spaces. The implementation of this integration includes the following transformations:

- The first group of transformations covers those transformations that approach abstract and concrete syntax of the same rule language, i.e., bridging different technical spaces:
  - The transformations between the XML and the MOF technological space, which are implemented by using the XML injector and extractor of the ATL toolkit. In particular, we developed two-way transformations between MOF-based and XML-based representations of the following languages:
    - SWRL XML concrete syntax (i.e., SWRL XML Schema that also includes OWL XML Schema) and Rule Definition Metamodel (RDM) as a MOF-based meta-model of the SWRL language (we name these transformations XML2RDM and RDM2XML);
    - R2ML XML Schema and the R2ML MOF-based meta-model (we name these transformations XML2R2ML and R2ML2XML).
  - The transformations between the EBNF and the MOF technological space which are implemented by using the EBNF injector and extractor of the ATL toolkit. These ATL's EBNF utilities are used for the transformation between the OCL concrete syntax and the OCL meta-model (i.e., OCL invariants and def's). We implemented this transformation by defining the OCL textual concrete syntax (TCS) and by generating the ANTLR based grammar, and thus generated OCL parser and lexer for bridging the gap between OCL textual concrete syntax and OCL MOF-based representation.
- The second group of transformations covers those transformations that are done in the same technological space (i.e., MOF). That is, these transformations bridge between the RDM, R2ML and OCL models (we name these transformations RDM2R2ML, R2ML2RDM, R2ML2OCL, and OCL2R2ML).

In this thesis, we decided to support the interchange of integrity rules, because SWRL rules and OCL invariant defines that type of rules. In the future work, we will support transformations between other rule languages (e.g., JBoss and JRules) and other type of rules (i.e., derivation, reaction, and production) with R2ML.

1.3. CONTENTS PER CHAPTER

This thesis contains seven chapters. After the introduction, chapter 2 overviews the research disciplines relevant to the research represented in this thesis. In that chapter, we first describe the notion of ontologies, as well as the basic ideas of the Semantic web. We also give an overview of languages used for representing and describing information resources and rules on the Semantic Web. Next, we defined the basic concepts of MDE, modeling, meta-models, and meta-modeling architectures. Then, we show basic concepts of model transformations (languages, definitions and applications). We also introduce the Model Driven Architecture, as one specific version of the MDE approach. This chapter ends with the concise overview of the Eclipse Modeling Framework conceptual environment for modeling, as well as with the defining the notions of modeling and technological spaces.

Chapter 3 introduces the R2ML rule markup language and its design and architecture aspects. The chapter starts with an overview of different types of rules that R2ML supports. After that, it shows the definitions of several rule languages, including, R2ML (with its XML Schema), RuleML, RDM, and OCL. Functional requirements that these meta-models support are also shown.

Chapter 4 presents the most important elements of mappings defined between rule language meta-models. This chapter starts with an overview of the conceptual solution for the transformation of Web rules and R2ML as a central rule markup language. Next, transformations are described (conceptually): between SWRL and R2ML with specific emphasis on the bridging XML and MOF technological spaces, where we define mappings and transformations between R2ML XML Schema and the R2ML model, between SWRL XML Schema and the RDM model, between the R2ML model and the RDM model, and between R2ML model and OCL model. This chapter concludes with the description of the transformations between the textual and MOF-based representations of the OCL language.

Chapter 5 gives in detail an overview of the transformations implemented. It briefly shows the architecture of the ATLAS development tools (i.e., ATL Editor) and how to use them. Next, we describe the implementation of transformations for integrity rules, between SWRL and OCL via R2ML. This chapter also illustrates models obtained in each step of the transformations by using both UML object diagrams and XMI representations. The chapter also presents a Java class that enables the execution of ATL transformations in regular Java-based applications (outside of the Eclipse-based development), and an architecture of a Java-based Web application for transforming rules that uses that class.

Chapter 6 is dedicated to the analysis of the important aspects of the design and implementation of the transformations for sharing rules between rule languages. First, we describe experiences in the development of the R2ML meta-model and transformations, and then discuss cons and pros of the transformations implemented. Finally, we compare the transformations proposed an implemented in this thesis with the existing and relevant solutions used for transforming Web rule languages.

Chapter 7 concludes this thesis by extracting conceptual and practical contributions of the thesis, by analyzing potentials of the practical use of the solution proposed, and by outlining the further research and development directions.

2. SURVEY OF RELEVANT TECHNOLOGIES

This chapter surveys the field and the technologies that are important for understanding the concepts described in the rest of this thesis. In addition, this chapter describes the Semantic Web, rules on the Semantic Web, the concepts of Model Driven Engineering, Eclipse Modeling Framework, as well as modeling and technological spaces.

2.1. SEMANTIC WEB AND RULES

This section describes the basic concepts of the Semantic Web, as well as rule languages used on the Semantic Web.

2.1.1. About The Semantic Web

Tim Berners-Lee, the creator of today's Web, has introduced the concept of the Semantic Web [Berners-Lee, 1998]. In his vision of the Semantic Web, data located somewhere on Web should be available, processable, and understood by both people and machines. The Semantic Web is a fluid, evolving, informally defined concept. It is an advanced phase of Web development, which should enable to overcome the problems that are identified on today's Web. In the first phase of its development, the Web was based on using HTML and was made completely of statically created pages. In the next phase, to enable the Web applications to use databases and dynamically generated Web content directly, various script languages have been developed. Examples of such languages are Common Gateway Interface (CGI), Active Server Pages (ASP), Java Server Pages (JSP), etc.

To enable semantic interpretation of data by machines, it is needed to separate the view from the content. In order to achieve this, some artificial intelligence techniques for knowledge representation have been proposed, as well as a wide spectrum of standards on which the Semantic Web is based. To get a feeling for this range of ideas, here are some representative quotations about the nature of the Semantic Web [Passin, 2004]:

- *The machine-readable-data view* - The Semantic Web is a vision: the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but also for automation, integration and reuse of data across various applications [W3C, 2003].
- *The intelligent agents view* - The aim of the Semantic Web is to make the present Web more machine-readable, in order to allow intelligent agents to retrieve and manipulate pertinent information [Cost et al., 2001].
- *The distributed database view* - The Semantic Web concept is to do for data what HTML did for textual information systems: to provide sufficient flexibility to be able to represent all databases and logic rules to link them together to great added value [W3C, 2000]. A simple description of the Semantic Web is that it is an attempt to do for machine processable data what the World Wide Web did for human readable documents. To turn the Web from a large hyperlinked book into a large interlinked database.
- *The automated infrastructure view* - Semantic Web is an infrastructure, and not an application [Berners-Lee et al., 2001].
- *The servant-of-humanity view* - The vision of the Semantic Web is to let computer software relieve us of much of the burden of locating resources on the Web that are relevant to our needs and extracting, integrating, and indexing the information contained within [Cranefield, 2001]. The Semantic Web is a vision of the next-generation Web, which enables Web applications to automatically collect web documents from diverse sources, integrate and process information, and interoperate with other applications in order to execute sophisticated tasks for humans [Anutariya, 2001].

- *The better-annotation view* - The idea of a 'Semantic Web' supplies the (informal) Web with annotations expressed in a machine-processable form and linked together [Euzenat, 2001].
- *The improved-searching view* - Soon it will be possible to access Web resources by content rather than just by keywords [Anutariya, 2001].
- *The Web services view* - Increasingly, the Semantic Web will be called upon to provide access not just to static documents that collect useful information, but also to services that provide useful behavior [Klein & Bernstein, 2001]. The Semantic Web promises to expand the services for the existing Web by enabling software agents to automate procedures currently performed manually and by introducing new applications that are infeasible today [Euzenat, 2001].

It follows from the above statements that the Semantic Web covers a wide range of areas and perhaps no two people have quite the same idea about it. However, some issues constantly dominate the research and development related to the Semantic Web [Passin, 2004]:

- indexing and retrieving information;
- meta data;
- annotation;
- the Web as a large, interoperable database;
- machine retrieval of data;
- Web-based services;
- discovery of services, and
- intelligent software agents.

To enable the Semantic Web to "live", it is necessary to provide a wide range of standards. Figure 2.1 shows the structure of the Semantic Web[3] which provides a framework for all such standards.

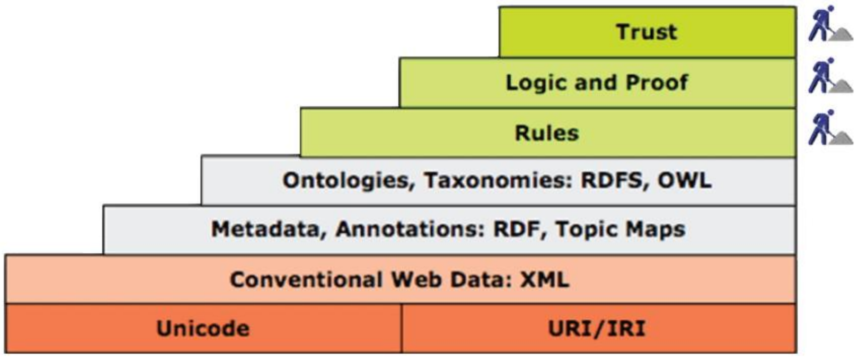

Trust
Logic and Proof
Rules
Ontologies, Taxonomies: RDFS, OWL
Metadata, Annotations: RDF, Topic Maps
Conventional Web Data: XML
Unicode
URI/IRI

Figure 2.1: Semantic Web architecture (adapted from [Berners-Lee et al., 2001])

Each layer is seen as building on the layer below. The enabling technologies of the Semantic Web are described later in this chapter.

2.1.2. Ontologies

The notion of ontologies in engineering refers to knowledge representation. If we consider intelligent systems that deal with representation and processing of knowledge, there is often the need to reuse the knowledge of some domain. It should be possible to reuse the knowledge collected while solving one problem, in new versions of the intelligent system. In addition, it is useful to enable domain knowledge to be used and exchanged by different users (humans or software agents). Ontologies enable

[3] Because of its specific shape, it is known as *The Semantic Web layer-cake*

such knowledge reuse and sharing, which is their main purpose [Gašević, 2004]. "*Ontology is an explicit specification of a conceptualization*" [Gruber, 1993] is the most often cited definition of an ontology.

The development of ontologies is much the same as the traditional approaches to development of software systems [Noy & McGuinness, 2001]. Some examples of such systems are object-oriented systems and databases. In essence, development of ontology consists of a domain analysis. Although software and ontology engineering processes have different aspects, convergence between those disciplines is more and more apparent. Because of that, the latest research suggests using software engineering techniques for ontology development (e.g., UML [Kogut et al., 2002], and software patterns [Devedžić, 2001]).

The structure of an ontology includes three important elements:

- taxonomy of the form *a-kind-of*, which represents a hierarchical organization of concepts;
- internal structure of concepts and their interconnections (i.e., *part-of*);
- explicit axioms, most frequently defined using some mathematical mechanism, such as first order logic, description logics, and conceptual graphs.

Specification of the ontology in knowledge systems have two dimensions: domain-factual knowledge and knowledge for solving problems [Chandrasekaran et al., 1999]. Kalfoglou ranks ontologies according to their purpose [Kalfoglou, 2001]. In the first group there are *ontologies for knowledge presentation*, and an example of such an ontology is the Frame ontology [Gruber, 1993], which specifies the primitives used in frame-based languages. In the second group are *task ontologies* which specify the knowledge of a task, which is independent from the domain in which that task can be performed [Mizoguchi et al., 1995]. *Method ontologies* complement task ontologies; they provide definitions of relevant concepts and relations that are used for defining reasoning processes.

When an ontology is created, it is necessary to define the following [Heflin, 2004]:

- classes in the domain of interest;
- relations which can exist between these classes;
- attributes that the classes can have;
- constraints on the attributes, which are used for consistency checking.

In the early research of ontologies, list-like syntax has been used for representing ontology knowledge [Jovanović, 2005]. An example of such a language is KIF (Knowledge Interchange Format). However, the appearance of eXtensible Markup Language (XML) as a specification for data interchange and interoperability on the Web, affected ontology languages in a way that those languages today have XML-based syntax.

2.1.3. Semantic Web languages

The first technology on which the Semantic Web is based is the standard language for data interchange *Extensible Markup Language* (*XML*). The basic idea related to the XML is to define a standard that will be used for data interchange on the Web, but with regard to the content, and not to the view. XML is a language that does not have a predefined set of keywords (elements and attributes), because it is a language created to define other languages, i.e. XML is a metalanguage. In addition, XML includes a few more technologies that it cannot be used efficiently without them [Hunter, 2001]:

- *XPointer* - enables addressing a certain part in an XML document;
- *XLink* – a standard for connecting XML documents;
- *XPath* – a standard that uses XPointer to specify paths to locations being addressed;
- *namespace* – a standard used to avoid name collisions;

- *eXtensible Stylesheet Language* (*XSL*) – a standard that consists of *XSL Transformations* (transform one XML document into another) and a language for formatting, *XSL formatting objects* (*XSL FO*).

In order to be correct, an XML model has to be *well formed* and *valid*. The first criterion implies compliance to general syntax rules of the XML, such as enclosing attribute values in quotation marks, or ensuring that all elements are balanced (each opened element must be closed). The second criterion means that the document must be written according some predefined specific grammar. To this end, two W3C standards are used: *Document Type Definition* (*DTD*) and *XML Schema*. DTD specifies how the elements of an XML document are defined, what are the attributes of an element, and what are its contents. DTD has many limitations [Roy & Ramanujan, 2001]. For example, it does not allow to define the number of elements that can be contained in another element. Likewise, there is no support for datatypes. Because of such limitations, the XML Schema standard was created. It overcomes DTD's problems, defines additional functionalities, 44 datatypes, and support for inheritance and precise definition of multiplicity. However, this is still not enough for representing ontologies, because semantic units for a specific domain cannot be recognized. In addition, it is very difficult to separate semantic relations between different concepts which exist in some domain, and it is difficult to create mapping between two domains, because both are defined with XML [Decker et al., 2000] [Klein et al., 2000]. To solve these problems, W3C consortium recommended RDF and RDFS languages.

Resource Description Framework (*RDF*) is a W3C standard [Lassila & Swick, 1999] created to standardize defining and using metadata, i.e. resource descriptions. RDF provides data model that supports a fast integration between data sources, thereby bridging semantic differences. As its name suggests, RDF is not a language but a model for representing data about things on the Web [Klein, 2001]. All elements that RDF describes are called resources. A resource can be anything that a URI can denote as a resource. The basic building block in RDF is the *object-attribute-value (O-A-V)* triple, which is often written as A(O,V). This means that some object O has attribute A with value V.

RDF Schema (*RDFS*) [Bickley & Guha, 2003] can be used to define vocabulary for RDF documents, and thus specify object types to which a certain property can be applied. This means that RDFS provides a basic typing mechanism for RDF models.

To enable reasoning services for the Semantic Web, another layer above RDF(S) is needed. That (logical) layer introduces ontological languages (see Figure 2.1), which are based on meta-modeling of the adjacent lower layer. It introduces a richer set of modeling primitives, which can be mapped to description logics. This enables using tools with generic support for reasoning that is independent of the problem domain. Examples of early languages of this kind are OIL and DAML. More recently, Web Ontology Language (OWL) is adopted by W3C as a standard ontology representation language for the Semantic Web.

Web Ontology Language (*OWL*) is a semantic language for publication and sharing of ontologies on the World Wide Web. OWL is developed by extending the RDF vocabulary and with the experience with DAML+OIL Web ontology language [Dean & Schreiber, 2004]. Since WWW is unlimited, OWL must start from the open-world assumption and enable inclusion and interference of different ontologies. Some of them might be contradictory, but new information must not overrun existing one, and may be only added. To enable such possibilities, and simultaneously support computing and reasoning in a finite time with tools that can be built with existing or forthcoming technologies, OWL introduces three sublanguages with different expressivity for different purposes:

- *OWL Lite* is OWL DL (see the next bullet point) with more restrictions. The idea is to make it easy to start with, and easy to implement processors, so that people can begin using OWL Lite easily and later graduate to more complex usage.
- *OWL DL* (*Description Logic*) enables maximum expressivity, and simultaneously ensures completeness of computation and decidability. Completeness means that all connections and

interlaces can be solved in finite time. A limitation is that, unlike OWL Full, OWL DL does not allow a class to be an individual and a property at the same time.
• *OWL Full* provides maximum expressivity and syntax divergence from RDF. The main characteristic of OWL Full, in contrast with OWL DL and OWL Lite, is that a class, which is defined as a collection of individuals, can be an individual itself, as in RDF(S).

2.1.4. Rule types

Rules are among the most frequently used knowledge representation techniques. There is a hierarchy of rule types, Figure 2.2, from *reaction rules* (event-condition-action rules), via *integrity rules* (rules for consistency checking) and *derivation rules* (implicitly derived rules), to *facts* (derivation rules without premises).

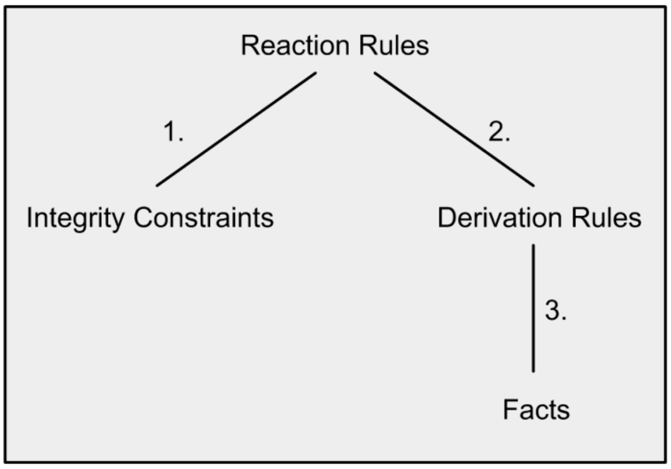


Figure 2.2: Hierarchy of rule types [Boley et al. 2001]

A reaction rule typically includes an event which triggers the execution of the rule, conditions that are necessary to execute an action that the rule defines, the action itself, as well as pre- and post-conditions. An example of such a rule is the trigger in the SQL language (CREATE TRIGGER expression), Figure 2.3. It is executed when data about buyers from the *buyers* table is changed or deleted, and that action is written into another table (*buyers_overview*).

```
CREATE TRIGGER before_buyer_change BEFORE UPDATE ON buyers
      FOR EACH ROW
      BEGIN
            INSERT INTO buyers_overview
            SET action='update',
                buyer_id = OLD.buyer_id,
                name = OLD.name,
                change = NOW();
      END
```

Figure 2.3: Creation of a trigger in SQL

An integrity rule consists of a logical sentence. Such rules specify statements that must be true in all states and state transitions of a discrete dynamic system for which they are defined. An example of such a rule is: "*Confirmation of a booking for a car with a car rental agency must take into account possibly existing bookings/assignments for the car, the requested car type, and the requested date*". Well-known integrity rule languages are SQL and OCL (discussed in section 2.2.1.6).

A derivation rule consists of one or more conditions and a conclusion that play an important role in a logical formula. An example of such a rule (expressed in a computation-independent way) is: "*A car is available for rental if it is not assigned to any client and is not scheduled for service*". An example of a language for representing derivation rules is RuleML (discussed in section 2.1.5.1).

2.1.5. Rule languages

The main purpose of a rule markup language is to permit reuse, interchange and publication of rules. Various rule markup languages are the vehicle for using rules on the Web and in other distributed systems [Wagner et al., 2006]. They allow deploying, executing, publishing and communicating rules in a network. In other words, rule markup languages allow us to specify business rules as modular, stand-alone units in a declarative way, and to publish and interchange them between different systems and tools. They play an important role in facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the Internet.

2.1.5.1. RuleML

RuleML represents an initiative for creating general rule markup language that will support different type of rules and different semantics [Boley et al., 2001]. It is conceptualized to capture the hierarchy of rule types shown in Figure 2.2. However, the current version of RuleML (February 2006) is 0.91 and it covers only some limited forms of rules. For example, in version 0.91 it still does not have a general syntax for integrity and reaction rules [Wagner et al., 2005].

RuleML is built on logic programming paradigm of first order logic (i.e., predicate logic), although they have similarities. In the tradition of logic programming that follows RuleML, research is focused on computable interpretations of predicate logic, by exploring a great number of semantic extensions and variations. OWL (as well as SWRL, see the next section) stems from logic-based tradition of artificial intelligence, where research is based on classic predicate logic (two-valued) as one and only logic.

An example of a RuleML rule that uses a certain person's attributes "hasMother" and "hasBrother", which implies the attribute "hasUncle" is shown in Figure 2.4.

```
<Implies>
      <head>
            <Atom>
                  <Rel>hasMother</Rel>
                  <Var>x1</Var>
                  <Var>x2</Var>
            </Atom>
            <Atom>
                  <Rel>hasBrother</Rel>
                  <Var>x2</Var>
                  <Var>x3</Var>
            </Atom>
       </head>
      <body>
            <Atom>
                  <Rel>hasUncle</Rel>
                  <Var>x1</Var>
                  <Var>x3</Var>
            </Atom>
      </body>
</Implies>
```

Figure 2.4: A rule in RuleML

Table 2.1 shows basic tags in RuleML, as well as their descriptions.

Table 2.1: Basic RuleML tags

Tag	**Description**
`<Implies>`	An implication rule. It consists of a conclusion role (`<head>`) followed by a premise role (`<body>`), or, equivalently (since roles constitute unordered elements), a premise role followed by a conclusion role.
`<Atom>`	A logical atom, i.e. an expression formed from a predicate (or relation) applied to a collection of its (logical) arguments.
`<Rel>`	A relation, i.e., a logical predicate, of an atom (`<Atom>`).
`<Var>`	A logical variable, as in logic programming.
`<Ind>`	An individual constant, as in predicate logic, which can also be considered to be a fixed argument like RDF resources.
`<op>`	An operator expression including either a relation (`<Rel>`) of an atom (`<Atom>`), a function name (`<Fun>`) of a (`<Expr>`), or a neutralized constant (`<Con>`) of a Hilog term (`<Hterm>`).

2.1.5.2. Semantic Web Rule Language (SWRL)

The most recent initiative in the W3C consortium for rule languages is Semantic Web Rule Language (SWRL), which actually represents a combination of the OWL and RuleML languages [Horrocks et al., 2004]. Like RDF and OWL, SWRL is based on classical first order logic. SWRL is restricted to OWL DL expressions, so it does not have support for high-level expressions that are allowed in RDF and OWL Full. This language is very similar to RuleML, and its rules are of the form of an implication between an antecedent (body) and a consequent (head). The intended meaning can be read as "whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold". Both the antecedent (body) and consequent (head) consist of zero or more atoms. Multiple atoms are connected with the conjuction operator.

Atoms in these rules can be of the form `C(x)`, `P(x,y)`, `sameAs(x,y)` or `differentFrom(x,y)`, where `C` is an OWL description, `P` is an OWL property, and `x`,`y` are either variables, OWL individuals or OWL data values. An example of such a rule for the implication *hasUncle* is shown in Figure 2.5.

```xml
<ruleml:Implies>
 <ruleml:body>
   <swrlx:individualPropertyAtom swrlx:property="hasMother">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x2</ruleml:Var>
   </swrlx:individualPropertyAtom>
   <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:Var>x2</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
   </swrlx:individualPropertyAtom>
 </ruleml:body>
 <ruleml:head>
   <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
   </swrlx:individualPropertyAtom>
 </ruleml:head>
</ruleml:Implies>
```

Figure 2.5: An example of the SWRL rule

2.1.5.3. General Web Rule Language (REWERSE I1 Rule Markup Language)

General rule markup language (GRML) [Wagner et al., 2005] is defined by its *abstract syntax* and *formal semantics*. Abstract syntax defines the basic categories of GRML, such as conditions, conclusions, events, actions and post conditions, as well as their subcategories - negation, conjunction, etc. This language also comes with formal semantics for rules: for logic expressions that can take the role of integrity constraints, preconditions, conclusion and post-conditions, for derivation rules, for events and actions, and for production and reaction rules. An example of the abstract syntax for derivation rules is shown in Figure 2.6.

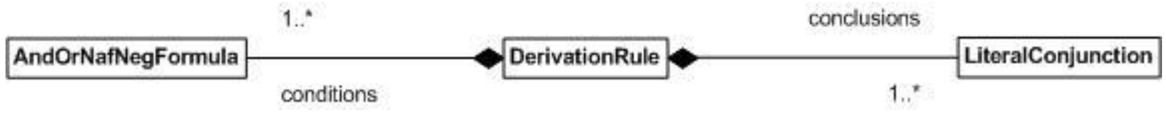


Figure 2.6: GRML abstract syntax for derivation rules

Formal semantics of GRML derivation rules is defined as a Tarski-style theory in [Wagner et al., 2005]. This theory includes the triple $\langle L, \mathbf{I}, \models\rangle$, such that:

- L is a set of formulae, called *language*;
- $\mathbf{I}$ is a set of interpretations;
- $\models$ is a relation between interpretations and formulae, called *model relation*.

For each Tarski-style model theory $\langle L, \mathbf{I}, \models\rangle$, one can define:

- a notion of a derivation rule $F \rightarrow G$, where $F \in L$ is called a "condition" and $G \in L$ is called a "conclusion".
- $\mathrm{DR}_L = \{ F \rightarrow G : F, G \in L \}$, the set of derivation rules of L;
- a standard model operator:

$$\mathrm{Mod}(X) = \{I \in \mathbf{I} : I \models Q \text{ for all } Q \in X\}$$

where $X \subseteq L \cup \mathrm{DR}_L$ is a set of formulae and/or derivation rules, and is called a *knowledge base*.

Typically, in knowledge representation theories not all models of a knowledge base are *intended* models. Except from the standard model operator *Mod*, there are also non-standard model operators, which do not provide all models of a knowledge base, but only a specific subset that is supposed to capture its intended models according to some semantics.

A particularly important type of such an ‘intended model semantics’ is obtained on the basis of some *information ordering* $\leq$, which allows comparison of the information content of two fact sets X_1, $X_2 \subseteq L$: whenever $X_1 \leq X_2$, we say that X_2 *is more informative* than X_1. Tarski-style model theory is defined as extended by an information ordering as a quadruple $\langle L, \leq, \mathbf{I}, \models\rangle$, and is called an *information model theory*.

For any information model theory, we can define a number of natural nonstandard model operators, such as the *minimal* model operator:

$$\mathrm{Mod}_m(X) = \{I \in \mathrm{Mod}(X) : I \leq I' \text{ for all } I' \in \mathrm{Mod}(X)\}$$

For any given model operator $\mathbf{M} : L \cup \mathrm{DR}_L \rightarrow \mathbf{I}$, and knowledge base $X \subseteq L \cup \mathrm{DR}_L$, we can define an entailment relation:

$$X \models_{\mathbf{M}} Q \text{ if and only if } \mathbf{M}(X) \subseteq \mathbf{M}(\{Q\})$$

For non-standard model operators, like minimal and stable models, this entailment relation is typically *nonmonotonic* in the sense that for an extension $X' \geq X$ it may be the case that X entails Q, but X' does not entail Q.

A concrete implementation of a general rule markup language is proposed in [Wagner et al., 2006], in the form of the rule markup language called REWERSE I1 Rule Markup Language (R2ML), and it is developed as part of the EC-funded REWERSE project. The R2ML language itself includes:

- datatypes and user-defined vocabulary;
- individual expressions - object and data expressions;
- atoms - basic parts of every rule;
- formulas - concepts and and-or-naf-neg formulas;
- actions - creation, deletion, assignment and calling;
- rules - integrity rules, derivation rules, and production rules.

The R2ML language represents a general format for rule interchange, instantiated in concrete rule languages for the Semantic Web, i.e., RuleML and SWRL with OCL. This language is semantically rich enough to permit preservation of constitutive structure of different languages and it does not enforce users to translate their rule expressions to different languages.

A detailed discussion of the general rule markup language can be found in [Wagner et al., 2005]. Elements of this language are discussed in Chapter 3.

2.2. MODEL DRIVEN ENGINEERING

The concept of Model Driven Engineering (MDE) appeared as a generalization of Model Driven Architecture (MDA), which represents a modern approach to software development [Kent, 2002]. This section presents the basic concepts of MDA and MDE.

2.2.1. Model Driven Architecture (MDA)

Model Driven Architecture (MDA) defines an approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform [Miller & Mukerji, 2003]. The MDA approach and the standards that support it enable a model that determines some system functionality to be realized on multiple platforms through additional standards for mapping. MDA is specified by the OMG consortium[4] in a series of standards: Unified Modeling Language (UML), Meta-Object Facility (MOF), Common Warehouse Metamodel (CWM), etc. An illustration of the MDA idea is shown in Figure 2.7.

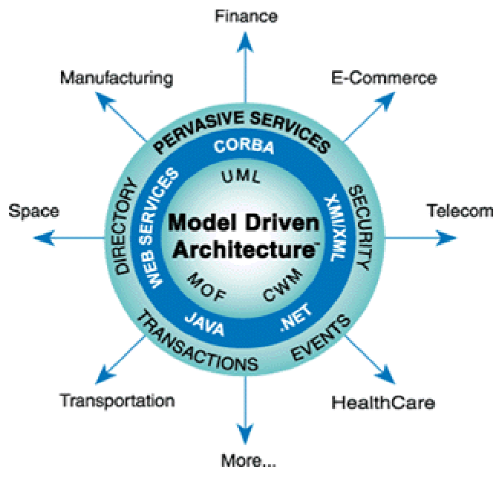


Figure 2.7: Model Driven Architecture (OMG)

[4] The Object Management Group, http://www.omg.org/.

2.2.1.1. Model Driven Architecture principles

Model is the most basic element of MDA. There are several definitions of the term "model" (see section 2.2.2.1), and the most general one is that a model is a simplified view of reality [Selic, 2003]. Each model itself is defined for some domain, and then it is transformed to models that can be executed on a specific platform.

A basic assumption of MDA is that a unique model underlies each information system. Such a model does not depend on a potential implementation platform, on which the corresponding application can be run. In other words, the system requirements can be specified as a *Computation Independent Model* (CIM) [Miller & Mukerji, 2003]. The model defined at this level is sometimes also called the *domain model* or the *business model*. It does not depend on how the system is implemented. In software engineering, a domain model is specified by the domain experts. This is very similar to the concept of ontologies (see section 2.1.2).

Platform Independent Model (PIM) can be also used to describe a system. It is lower-level and more specific than CIM in terms of being a computation-related model, but it does not include characteristics of specific computer platforms.

To get a model that takes into account some target platform specifics, i.e., a *Platform Specific Model* (PSM), it is needed to define certain transformations that transform the corresponding PIM to the desired PSM. Each PSM includes information about some software implementation details (such as the programming language and operating system) and the hardware platform. Code generation is done by additional translation from the PSM into a certain programming language.

MDA is based on four-layer metamodeling architecture shown in Figure 2.8. The standards supporting the four-layer MDA architecture are:

- Meta-Object Facility (MOF);
- Unified Modeling Language (UML);
- XML Metadata Interchange (XMI).

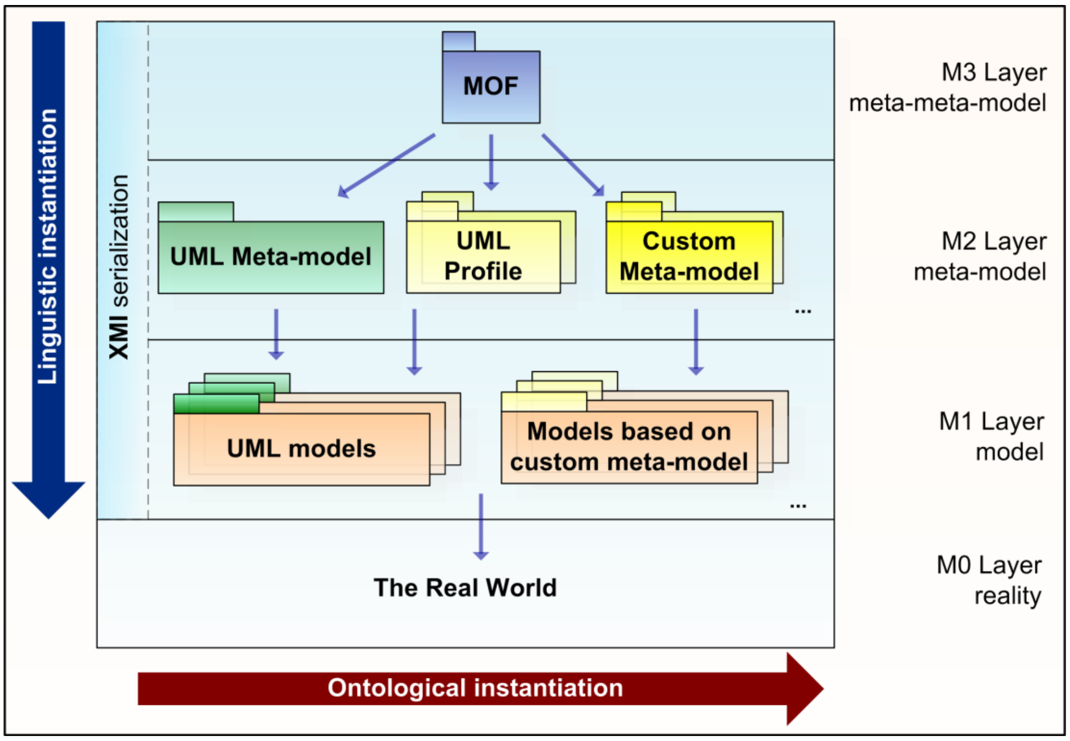


Figure 2.8: The four-layer Model Driven Architecture and its orthogonal *instanceOf* relations: linguistic and ontological [Gašević et al., 2006]

On top of this architecture, at the M3 level, is a reflexive meta-metamodel, which is called MOF. It is an abstract self-defined language and a framework for specifying, constructing, and managing technologically independent meta-models. It is a basis for defining any modeling language, such as UML or MOF itself. MOF also defines a backbone for the implementation of a metadata (i.e., model) repository described by meta-models. The rationale for having these four levels with one common meta-metamodel is to enable both the use and generic managing of many models and meta-models, and to support their extensibility and integration.

All meta-models, standard and custom (user-defined), that are defined in MOF are placed at the M2 level. One of these meta-models is UML, which is a language for specifying, visualizing, and documenting software systems. The basic UML concepts (e.g. Class, Association, etc) can be extended in UML profiles in order to adapt UML for specific needs. Models of the real world, which are represented by concepts of a meta-model from the M2 level, are at the M1 level of the MDA four-level architecture. The bottom layer is the instance layer (M0). At the M0 level are things from the real world that are modeled at the M1 level. For example, the MOF Class concept (from the M3 level) can be used for defining the UML Class concept (M2), which further defines the Student concept (M1). The Student concept is an abstraction of a real thing *student*.

One can ask the question: what layer contains abstractions of a certain model? If we consider classes, their instances in UML are objects. However, objects are defined at the M2 level in the UML meta-model, which means that their instances are located in the M1 layer. Since even objects themselves model concrete (singular) real-world things, this explanation can be considered true. In [Atkinson & Kühne, 2003] it is said that there are two types of instantiation in meta-modeling: *linguistic* and *ontological*. Linguistic instantiation is interpretated in MDA in an ordinary way - it means that a UML class is an instance of the meta-class from the UML meta-model. However, one class in some domain has instances that are objects. The relation between objects and classs is an ontological instantination relation. This kind of instantination connects abstractions located in the same linguistic layer. According to this interpretation, in the M0 layer are things from real world (instances) and abstract concepts about thing groups (classes). UML 2.0 and MOF 2.0 emphasize linguistic dimension. Ontological levels exist at the M1 level, but the meta-model border does not explicitly separate them. This is based on an altered perception of the MDA four-layer architecture, because originally class instances have been located in the M0 layer.

XML Metadata Interchange (XMI) is the standard that defines mappings of MDA-based meta-metamodels, meta-models, and models onto XML documents and XML Schemas [XMI2, 2005]. Since XML is widely supported by many software tools, it empowers XMI to enable better exchange of meta-metamodels, models, and models (see section 2.2.1.5).

2.2.1.2. Meta-Object Facility (MOF)

Meta-Object Facility (MOF) [MOF, 2005] in its current version (2.0) represents an adaptation of the UML core. MOF is a minimal set of concepts that can be used to define other modeling languages. It is similar (but not identical) to the part of UML used in structural modeling. In the latest version of MOF (2.0), concepts, as well as UML Superstructure concepts [UML, 2005], are derived from the concepts defined in the UML Infrastructure standard [UML, 2004].

Figure 2.9 shows meta-models that depend on the UML core package. UML Core package defines the basic concepts that are used in modeling, e.g. Elements, Relationships, Classifiers, etc. In MOF 2.0, there are two meta-metamodels:

- *Essential MOF* (EMOF) - represents a basic package that has a minimal number of elements for modeling, e.g. `Class`, `Property`, `Operation`, etc.
- *Complete MOF* (CMOF) - more complex, includes EMOF, but also enables a higher expressivity, with concepts such as `Link`, `Argument`, `Extent`, `Factory`, etc.

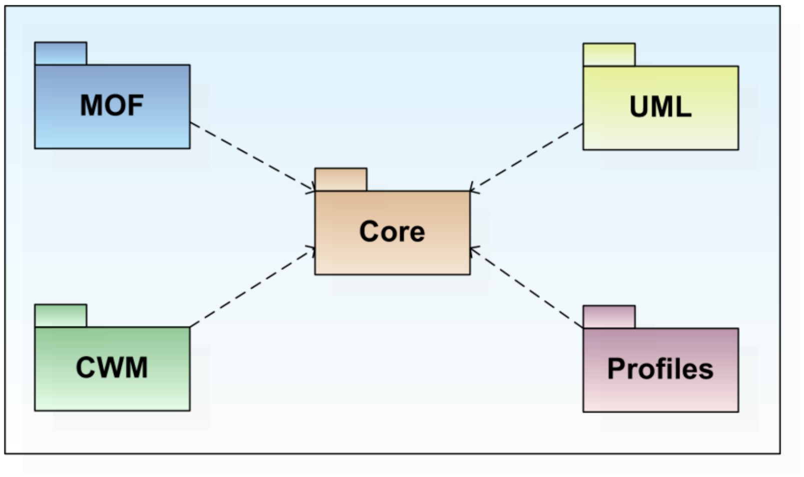


Figure 2.9: Core package as the common kernel [Gašević et al., 2006]

The main four modeling concepts in MOF are:

- `Class` - models MOF meta-objects, concepts which are entities in meta-models (i.e. UML `Class`, `Attribute` and `Association`, ODM `Class` and `Property`, etc.);
- `Association` - models binary relationships (UML and MOF superclass, for example);
- `Package` - modularizes other concepts, i.e. groups similar concepts;
- `DataType` - models primitive types (`String`, `Integer`, etc.).

In the root of the MOF hierarchy is the `Element` concept. It classifies elementary, atomic model elements. All other concepts in MOF inherit from this concept.

2.2.1.3. Unified Modeling Language (UML)

Unified Modeling Language (UML) is a language for specifying, visualizing, and documenting software systems, as well as for modeling business and other non-software systems [UML, 2005]. UML enables diagram construction, which models a system by describing conceptual things (e.g., a business process) and concrete thigns (e.g., software components). UML is not limited only to software engineering domain; it can be used in other areas: banking, health care, defense, etc. UML is often identified as a graphical notation, which was true for its initial versions. Recently, UML is recognized more like a language independent from a graphical notation rather than a graphical notation itself.

The basic building block of UML is a diagram. There are several types of diagrams for specific purposes (e.g., time diagrams) and a few for generic use (e.g., class diagrams). UML version 2.0 defines the following types of diagrams:

- use case diagram;
- class diagram;
- behavior diagrams:
  - activity aiagram;
  - statechart diagram;
- interaction diagrams:
  - sequence diagram;
  - collaboration diagram;
- implementation diagram;
  - component diagram;
  - deployment diagram.

When UML is applied to software, it represents a bridge between the original idea for some software and its implementation [Pilone & Pitman, 2005]. UML also provides a possibility for collecting specific requirements for some specific system.

UML as a graphical notation is not a software process; it is designed for use in a process of software development and it possesses all characteristics that enable it to be a part of a software development process. Since main UML diagram concepts are defined in the *Superstructure* package of the UML specification that includes basic concepts of the UML core [UML, 2005], it can be said that MOF and UML are very similar.

2.2.1.4. UML Profiles

UML Profiles combine concept *stereotypes*, *tagged values*, and *constraints* in order to define a precise UML dialect for a specific purpose. This means that it is possible to create new types of elements for modeling by extending existing elements. When new elements are created, it is possible to add them to existing UML tools. With profiles, classes can be extended with stereotypes that represent predefined classes with certain methods and attributes. For example, Figure 2.10 shows one such a stereotype - EJBEntityBean.

A UML Profile definition in the context of the MDA four-layer meta-modeling architecture means extending UML at the meta-model layer (M2). Tagged values are defined as stereotype attributes (in Figure 2.10 tagged values of `EJBEntityBean` are *IsReadOnly*, *DataSource*, etc.). It is possible to define constraints that additionally refine the semantics of the modeling element they are attached to. They can be attached to each stereotype using OCL (Object Constraint Language) or English language (i.e. spoken language) comments, in order to precisely define the stereotype's semantics.

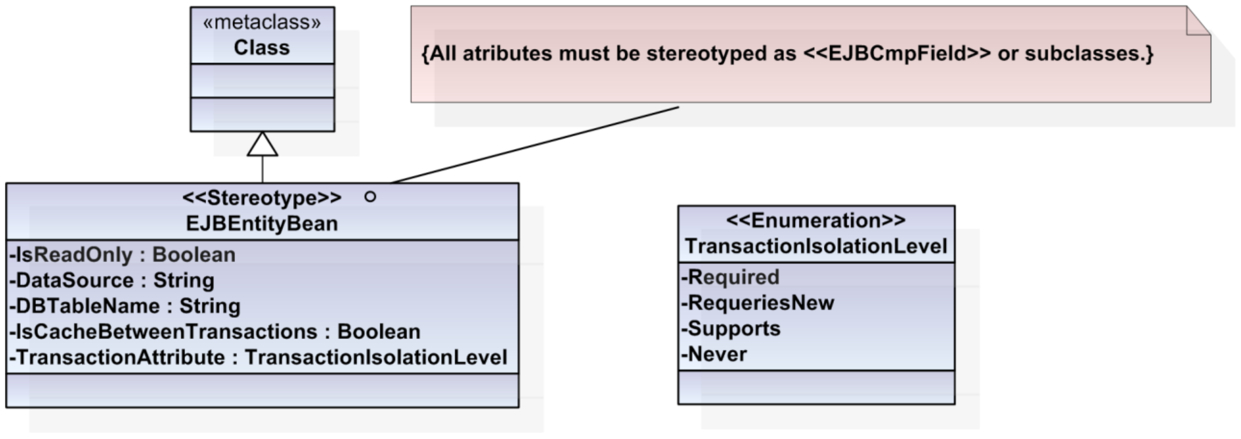

«metaclass»
Class
{All atributes must be stereotyped as <<EJBCmpField>> or subclasses.}
<<Stereotype>>
EJBEntityBean
-IsReadOnly : Boolean
-DataSource : String
-DBTableName : String
-IsCacheBetweenTransactions : Boolean
-TransactionAttribute : TransactionIsolationLevel
<<Enumeration>>
TransactionIsolationLevel
-Required
-RequeriesNew
-Supports
-Never

Figure 2.10: An example UML Profile for Enterprise applications in Java

So far, many important UML Profiles have been developed. Some UML Profiles are adopted by OMG, such as Enterprise Application Integration [EAI, 2004] and UML Profile for MOF [MOF, 2004]. In addition to these formal specifications, there are several well-known UML Profiles widely accepted by software engineers. One of the most popular ones is the UML Profile for building Web applications developed by Jim Conallen [Conallen, 2002].

2.2.1.5. XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is an XML-based standard for sharing meta-data in the MDA architecture [XMI2, 2005]. XMI is defined by XML, using two XML Schemas:

- XML Schema for MOF meta-models;
- XML Schema for UML models.

The first one defines the syntax for sharing both MOF-based meta-models and the MOF definition itself. Since UML is a modeling language that developers use for describing various models, it is obvious that there is a need for an XML Schema for exchanging UML models. In fact, there is a standardized one called the UML XMI Schema. The UML tools such as IBM/Rational Rose, Poseidon for UML, Together, etc. support it, but some researchers report that we always loose some information when sharing UML models between two UML tools [Uhl & Ambler, 2003]. OMG has released several versions of the XMI standard: 1.0, 1.1, 1.2 and 2.0, and the latest version is 2.1.

Figure 2.11. shows the relationship between UML models and XMI files.

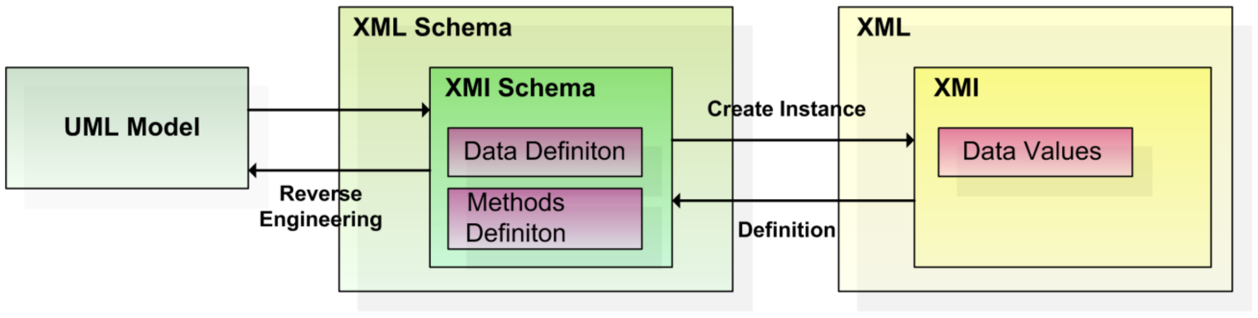


Figure 2.11: Relationship between UML, XML Schema and XMI

Since there is a set of rules for mapping UML and MOF models to XML Schema, it is possible to create XML Schema for every UML model. Objects as instances of such a model can be interchanged conforming to these schemas. An XML Schema can be created for any MOF-based meta-model.

An example of an XMI file (in version 1.2) is shown in Figure 2.12.

```
<XMI xmi.version = '1.2' xmlns:Model = 'org.omg.xmi.namespace.Model'>
 <XMI.content>
   <Model:Package xmi.id = 'a1' name = 'OCL' annotation = '' isRoot = 'false'
     isLeaf = 'false' isAbstract = 'false' visibility = 'public_vis'>
     <Model:Namespace.contents>
       <Model:Association xmi.id = 'a2'
         name = 'A_Operation_parameters_Parameter_operation'
         annotation = '' isRoot = 'true' isLeaf = 'true' isAbstract = 'false'
         isDerived = 'false'>
         <Model:Namespace.contents>
           <Model:AssociationEnd xmi.id = 'a3' name = 'parameters' annotation = ''
             isNavigable = 'true' aggregation = 'none' isChangeable = 'true'>
             <Model:AssociationEnd.multiplicity>
               <XMI.field>0</XMI.field>
               <XMI.field>-1</XMI.field>
               <XMI.field>true</XMI.field>
               <XMI.field>true</XMI.field>
             </Model:AssociationEnd.multiplicity>
             <!--...-->
            </Model:AssociationEnd>
            <!--...-->
         </Model:Namespace.contents>
        <!--...-->
       </Model:Association>
      </Model:Namespace.contents>
    </Model:Package>
 </XMI.content>
</XMI>
```

Figure 2.12: An excerpt from the MOF XMI document representing the OCL meta-model

2.2.1.6. Object Constraint Language (OCL)

Object Constraint Language 2.0 (OCL) as an addition to the UML 2.0 specification. It provides a way for expressing constraints and logic in models. OCL represents a language for defining integrity rules. It is not new in UML 2.0; OCL was first introduced in UML 1.4. However, from UML version 2.0 it is formalysed by using Meta-Object Facility (MOF 2.0) and UML 2.0, which is defined in UML OCL2 specification [OCL, 2006]. The Object Constraint Language (OCL) is just what its name says: a language. It has its syntax and semantics defined by the UML language, and it also has keywords. However, in contrast to other languages, OCL can be used for expressing programming logic or flow control. By its design, OCL represents just a query language, and it cannot change a model in any way [Pilone & Pitman, 2005].

OCL can be used for expressing: different pre- and post-conditions, invariants (constraints that always must be true), constraint conditions, and results of model executing. It can be used anywhere in UML, and it is usually associated to a class by using a comment (annotation). When an OCL expression is evaluated, the result is temporary. This means that the associated class, i.e., its concrete instances (objects), cannot change its condition during the expression evaluation.

OCL has four basic data types: `Boolean`, `Integer`, `Real` and `String`. Each OCL expression must have a context. The context can often be identified by where the expression is written. For example, a constraint can be attached to an element by using a comment. The context of a class instance can be referred to by using the keyword `self`. For example, if we have a constraint on the class `Student` that says: "a student's average grade (attribute *average* of type `Real`), must always be greater than 5.0", an OCL expression can be attached to the class `Student` by using a comment and by referring to the average in this way: `self.average > 5.0`.

OCL also includes constraints on methods and attributes, as well as different types of conditions, and possesses a possibility (methods) for manipulating data collections.

2.2.1.7. Query View Transformation (QVT)

MOF Query/View/Transformations (QVT) is a language proposed by Object Management Group (OMG) for model transformations and manipulation [QVT, 2005]. QVT specification is in the core of MDA. It integrates the OCL 2.0 standard and extends it to imperative OCL. OCL is used for queries on models. Model creation and overview is not suggested in this proposal. Queries take a model as input, and choose elements from the model. Views are models that inherit other models. Transformations take a model as input, and change it or create a new model. By using QVT, models become useful elements in development, where various intermediary tools can be avoided.

QVT has its place in the MOF-based meta-modeling architecture - QVT abstract syntax is defined as a MOF 2.0 based meta-model. QVT architecture is shown in Figure 2.13. This meta-model defines three Domain Specific Languages (DSLs) for model transformations, named *Relations*, *Core* and *Operational Mappings*. These three languages form a hybrid transformation language: Relations and Core are declarative languages at two different levels of abstraction, with a normative mapping between them. The Relations language has a graphical concrete syntax and it specifies a transformation as a set of relations between models. The Operational Mapping language is an imperative language that extends both Relations and Core. The syntax of the Operational Mapping language provides constructs commonly found in imperative languages (loops, conditions, etc.). QVT has a mechanism called *BlackBox* for invoking transformation facilities expressed in other languages (*XSLT*, *XQuery*), which is also an important part of the specification. It is especially useful for integrating existing non-QVT libraries.

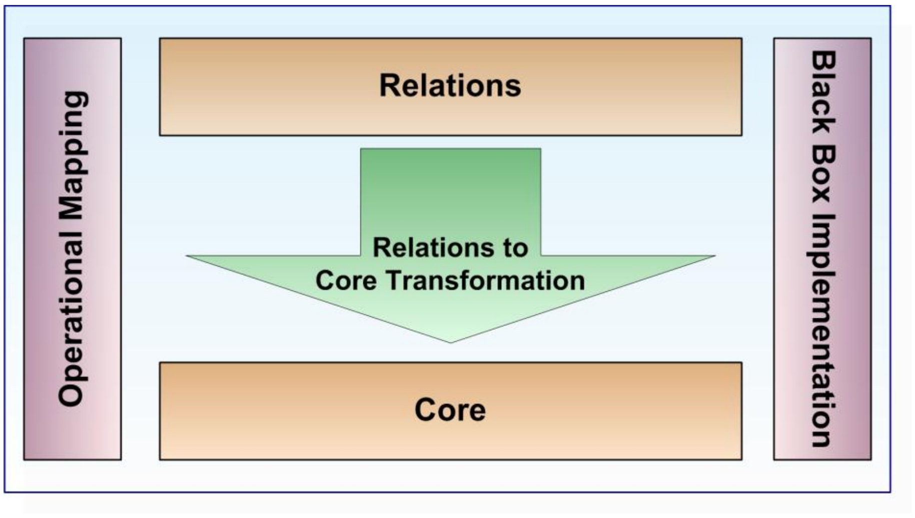


Figure 2.13: QVT architecture

QVT standard addresses only model-to-model transformations, where "model" is some entity conforming to any MOF 2.0 meta-model. All transformations of type *model-to-text* or *text-to-model*, whatever the text is (XML, Code, SQL, etc.), are presently outside the scope of QVT and possibly subject to other standardization initiatives. They may be viewed as alternative transformation DSLs in the MDA technological space (see section 2.4. for a detailed explanation of the concept of technological spaces).

2.2.1.8. Java Metadata Interface (JMI)

Java Metadata Interface (JMI) [Dirckze, 2002] defines translations of MOF-based meta-models to Java interfaces. These interfaces enable users to create, update or access meta-model instances by using Java programming language. Based on any MOF-based meta-model serialized to XMI, it is possible to generate JMI-based interfaces specific for that meta-model. These interfaces are used for accessing Java meta-data repository, which implements Java classes. For example, for the class `Class1` defined in a meta-model, two interfaces are generated: `Class1` that contains set/get methods, and `Class1Class` that contains a method that creates an object of this class.

All data from the repository can be serialized into XMI and interchanged with other repositories, regardless of the way in which the repositories are implemented. It is necessary only that the repositories comply with the MOF specification, e.g., to "understand" the MOF XMI format.

The main advantage of meta-data API is that it enables software to use objects without prior knowledge about them. In the MOF context, meta-object enables a program to "discover" semantics of any object, e.g., creation, updating, accessing, navigation and operation invoking on objects that are class proxies.

An example of JMI implementation is NetBeans Metadata Repository (MDR)[5] that implements MOF repository by including a mechanism for storing MOF-based meta-data.

JMI 1.0 specification is based on the MOF 1.4 specification [MOF, 2002], and is proven to be very complicated for implementation. Eclipse Modeling Framework (EMF) [Budinsky et al., 2003] is an alternative conceptual framework being developed in the Eclipse foundation, and has a simpler implementation (see the next section). EMF, just like JMI, defines translation of meta-model to Java API. These meta-models are not based on the MOF meta-metamodel, but on the ECore meta-metamodel, which is simpler for implementation. ECore is meta-metamodel in Eclipse Modeling Framework, and it is described in section 2.3.

[5] http://mdr.netbeans.org/

2.2.2. Basic concepts of Model Driven Engineering

Model Driven Engineering is not Model Driven Architecture [Favre, 2004]. MDA is an OMG standard and is a specific version of the MDE approach. Favre defines MDE as an open and integrative approach to software development which involves many technological spaces (TS) [Kurtev et al., 2002] in a uniform way, and MDA is only one instance of MDE implemented in a series of technologies defined by the OMG (MOF, UML, XMI).

MDA introduces a set of basic concepts, such as *model*, *meta-model*, *modeling language* and *transformation*, and recommends categorization of all models to platform-independent models (PIMs) and platform-specific models (PSMs). However, MDA is not a software development process.

A *technological space* is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities [Kurtev, 2002]. It is often associated with a given user community with shared expertise, educational support, common literature and even workshop and conference meetings. Examples of technological spaces are MDA and MOF, but also *Grammarware* [Klint et al., 2005] and BNF, *Documentware* and XML, *Dataware* and SQL, *Modelware* and UML, etc.

An important aspect of MDE is that it bridges different technological spaces and integrates knowledge from different research communities. In every space, model, meta-model and transformation concepts appear at various levels of abstraction and in a way can conform to certain concepts in another technical space. For example, what is called meta-model in Modelware, conforms to something that is called Schema in Documentware, grammar in Grammarware, etc. In [Kent, 2002], MDE is defined starting from MDA by adding assignment in a process of software development and a space for model organization. Two illustrative examples of the MDE process can be found in [Alanen et al., 2003] and [Bézivin et al., 2003].

2.2.2.1. Definitions of model and modeling

The origin of the word *model* can be traced to the Latin *modulus*, which means a small measure. A definition of model from [Starfield et al., 1990] says that: "*a model is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details*". Miller and Mukerji define model, as "*A model of a system is a description or specification of that system and its environment for some purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language*" [Miller & Mukerji, 2003].

Computer science uses models in several phases of software development. MDA and MDE rely on modeling and models as their basic concepts. However, there is no single definition of model that is widely accepted in all computer science. Seidewitz defines model as "*a set of statements about a system under study*" [Seidewitz, 2003], and [Kühne, 2006] defines model as an "*abstraction of (real or language-based) system allowing predictions or inferences to be made*". There is number of other definitions, presented in [Kurtev, 2005]. This thesis uses the following definition of model: "*A model represents a part of the reality called the object system, and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system*" [Kurtev, 2005].

Models usually serve as specifications in traditional engineering disciplines. When software is constructed, models can be used as specifications as well. A UML model can be used for describing an existing software system (its structure and operations).

Model interpretation means mapping model elements to the elements of the object system (system under study), so that a specific value of each model expression is obtained in the object system which is under study (with a certain level of accuracy). Thus, a model interpretation gives a model a meaning associated with the object system.

Modeling languages enable to write expressions with elements in models of classes systems under study. A working software system can be based on a model that represents a certain part of reality, while the software itself can be regarded as a model.

2.2.2.2. Modeling principles

In the world of software engineering, modeling has a rich tradition that reaches early days of programming. More recent efforts are focused on notation and tools that permit users to express the system parameters to software architects and programmers, in a way that can be uniquely mapped to a concrete programming language and then compiled for a specific operating system. UML [UML, 2005] is currently the most widely accepted language for visual specification of models, which is adopted as the de facto industry standard for software modeling and standardized by the Object Management Group (OMG). UML enables development teams to describe important characteristics of systems in appropriate models. Transformations between these models are usually accomplished manually, although there are tools that can do automatic model transformation [Metzger, 2005].

A model is used for an indirect study of reality (i.e., of an object system) [Kurtev, 2005]. Various reasons may cause this indirectness. The object system may be inaccessible, or its direct study is too expensive, or even the object system may not exist yet. In all such cases, the model plays the role of a specification of the object system. Regardless of the reasons for indirectness, the model must be a valid representation of the object system. The knowledge acquired from the model must hold for the object system. Often, this knowledge is not exact but only approximates the reality, with an acceptable degree of inaccuracy. Furthermore, the knowledge acquired from the model is initially expressed in terms of model elements. This knowledge must be interpreted and converted to knowledge in terms of the object system. The relation between a model and an object system is bi-directional and two separate relations may be considered, as Figure 2.14 shows. This figure is called the *DDI account* (DDI – Denotation, Demonstration, Interpretation), and was first introduced in [Hughes, 1999].

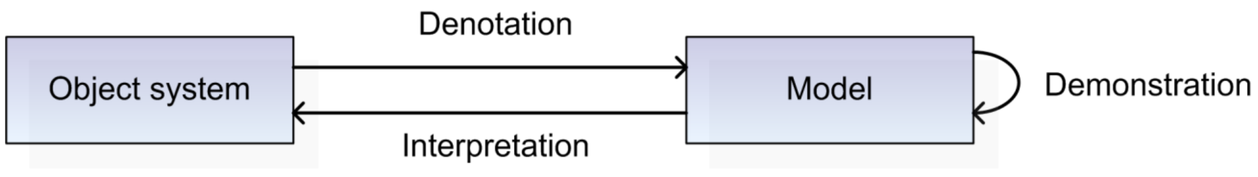


Figure 2.14: Relationships between an object system and its model [Hughes, 1999]

The object system is *denoted* (represented) in a model. This denotation must preserve some characteristics of the object system to allow acquiring knowledge about it through the model. The model is used to obtain claims about the model elements. This process is known as *demonstration*. It happens only in the context of the model. Finally, the obtained results are mapped to the object system. This mapping is called *interpretation*. The knowledge obtained from the model must be verifiable against the object system. If the results obtained from the model do not meet the empirical evidence obtained from the reality, then the model is invalid with respect to the object system.

Literature usually depicts only one relation between a model and its object system. Various names for the relation are used: *ModelOf, RepresentationOf, RepresentedIn, ModeledBy,* etc. *ModelOf* relation will be used in the remaining part of the thesis, because it accumulates two other relations: *Denotation* and *Interpretation*.

2.2.2.3. Meta-models and meta-modeling

As the name suggests, meta-modeling is a modeling activity. Similarly, the product of meta-modeling, called a *meta-model*, is a model. If an entity is a model, we have to be able to clearly identify its object system. Meta-model is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules determining the set of possible models denotable in that language [Falkenberg, et al., 1998]. Therefore, a meta-model describes what models in that language can express. Based on this, we can conclude that a meta-model is a model of models expressed in a given modeling language [Seidewitz, 2003]. Since a meta-model itself is a model, it is also represented in some modeling language. One modeling language can have more than one meta-model, each one represented in a different modeling language. Of special interest is the case when the meta-model of a modeling language uses the same modeling language. In that case, expressions in the meta-model are represented in the same language that describes the meta-model. This meta-model is called *reflexive meta-model*. *Minimally reflexive meta-model* uses a minimum number of modeling language elements (for that meta-model purpose). Since this meta-model is defined as reflexive, there is no need for upper levels, because it defines itself with its own concepts.

Generally, there is a *modelOf* relation between a meta-model and its object system; it is a modeling language. *InstanceOf* relation between a meta-model and a model often replaces it. Indeed, they coincide between the same entities but are different in nature. The grammar of some programming language possesses characteristics of all words (and sentences) which that language can contain. So we can take a language grammar as a model of that language (an example of such a grammar is the Extended Backus-Naur Form, EBNF). In the case of a modeling language, the model of this language is its meta-model. The relation between a model written in some language and its meta-model is called *conformantTo* [Favre, 2004-1]. This relation is defined as a composition of two relations: *elementOf*, denoting the membership of a model to a language, and *representationOf*, denoting the relation between an object system and its model. An example of a meta-model, a model, and an *instanceOf* relation is shown in Figure 2.15.

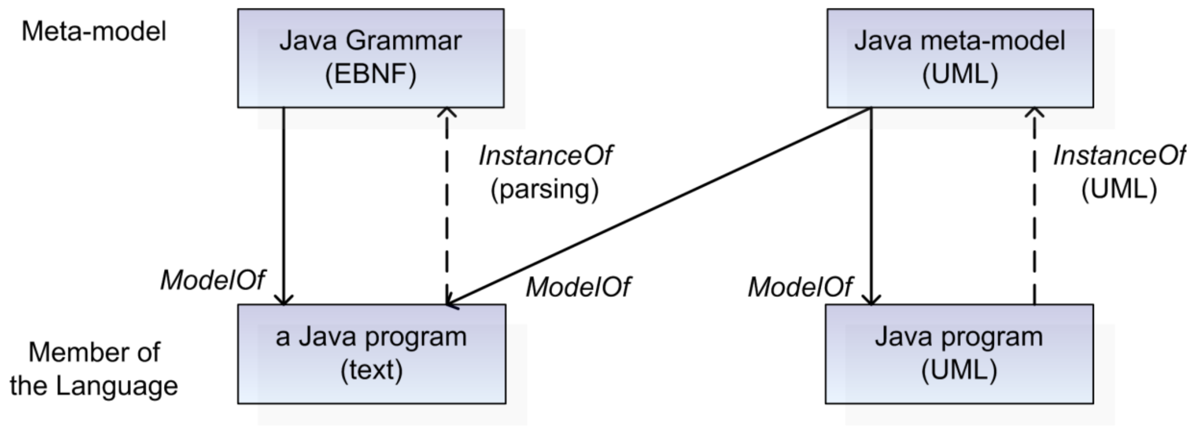


Figure 2.15: Example of meta-models, models and *instanceOf* relations [Kurtev, 2005]

An important difference between the two relations is observed when the language-dependent nature of *instanceOf* is considered. Let us assume that we define another meta-model of Java language expressed in UML (see Figure 2.15). The UML meta-model may contain a class called `Method`. The knowledge we obtain is that there is a set of methods in every Java program that has a certain structure. We must be able to identify methods in the source program and to recognize their structure according to the definition of the `Method` class. It is the consequence of the *ModelOf* relation that exists between the Java meta-model and a Java program. However, we cannot consider the Java program as an instance of the UML model in the same way as we did it for the Java grammar. An instance of the UML model is defined according to the semantics of UML, and is a set of objects. This instance is a representation of the Java program and is a different entity. The UML model of Java is also a model of the Java program represented in UML. In addition, there is an *instanceOf* relation between these entities governed by the UML semantics. Much like the relation between a source program and its grammar, this *instanceOf* relation helps us interpret the knowledge from the UML model in terms of the Java program represented in UML. These two *instanceOf* relations are different. The first one is defined for the parsing process. The second one relies on the UML semantics. There is no direct language-specific *instanceOf* relation between a source program in Java and its UML model. However, the latter is a model of the former, although we cannot trace the knowledge from the model to the object system via an *instanceOf* relation.

In summary, we can say that *instanceOf* relation exists between a class and its members and supports the interpretation of the knowledge obtained from the class definition in terms of class members. In that case, we also have a *ModelOf* relation between the class definition and class members.

2.2.2.4. Meta-modeling architecture

Meta-modeling activity can be applied to specify a modeling hierarchy that assumes a multi-level organization, called *meta-modeling architecture*. Figure 2.16 shows an example of this architecture.

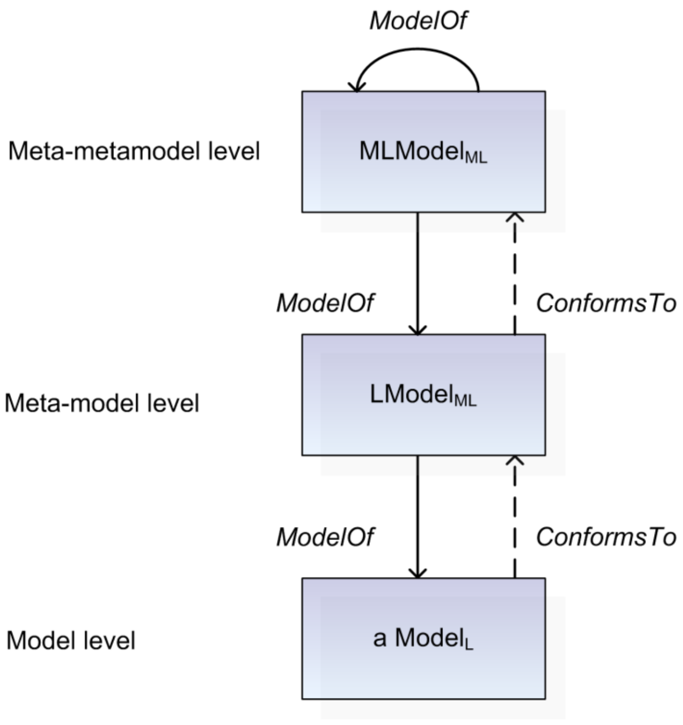


Figure 2.16: Meta-modeling architecture [Kurtev, 2005]

The *ConformsTo* relation is another name for *InstanceOf* relation, and it means that a model is constrained by the rules defined in its meta-model. At the bottom level of this architecture, we have models expressed in various modeling languages. This level is called the *model level*. An example model in this level is $Model_L$ written in a modeling language *L*. We can build a model of *L* (that is, a meta-model) $LModel_{ML}$ expressed in another language, called *Meta-language* (ML). Models of the languages used in the model level form the second level in the stack. It is called the *meta-model level*. There is a *ModelOf* relation between the meta-model of a language and models expressed in that language. We can apply the same approach to the models at the meta-model level. The models of the languages that express meta-models form the third level, called the *meta-metamodel level*. At the third level of the meta-modeling architecture shown in Figure 2.16, the model *MLModel* is expressed in the *ML* language itself. In this way, the top level contains a self-reflective model. It is expressed in the language that is modeled by that model. The intuition behind this is the following. At the meta-model level, we have models of modeling languages expressed in *ML*. However, *ML* is a modeling language itself, and therefore it should be possible to apply *ML* itself to express its model.

Examples of technologies that rely on meta-modeling architecture are Meta Object Facility (MOF), section 2.2.1.2, and Eclipse Modeling Framework (EMF), section 2.3.

An example of the relation between a model and its meta-model in Figure 2.17 that represents the *meta* relations between a Petri Net model and a simplified Petri Net meta-model, represented in UML. *Meta* relation, associates each element of a model with the meta-model element it instantiates.

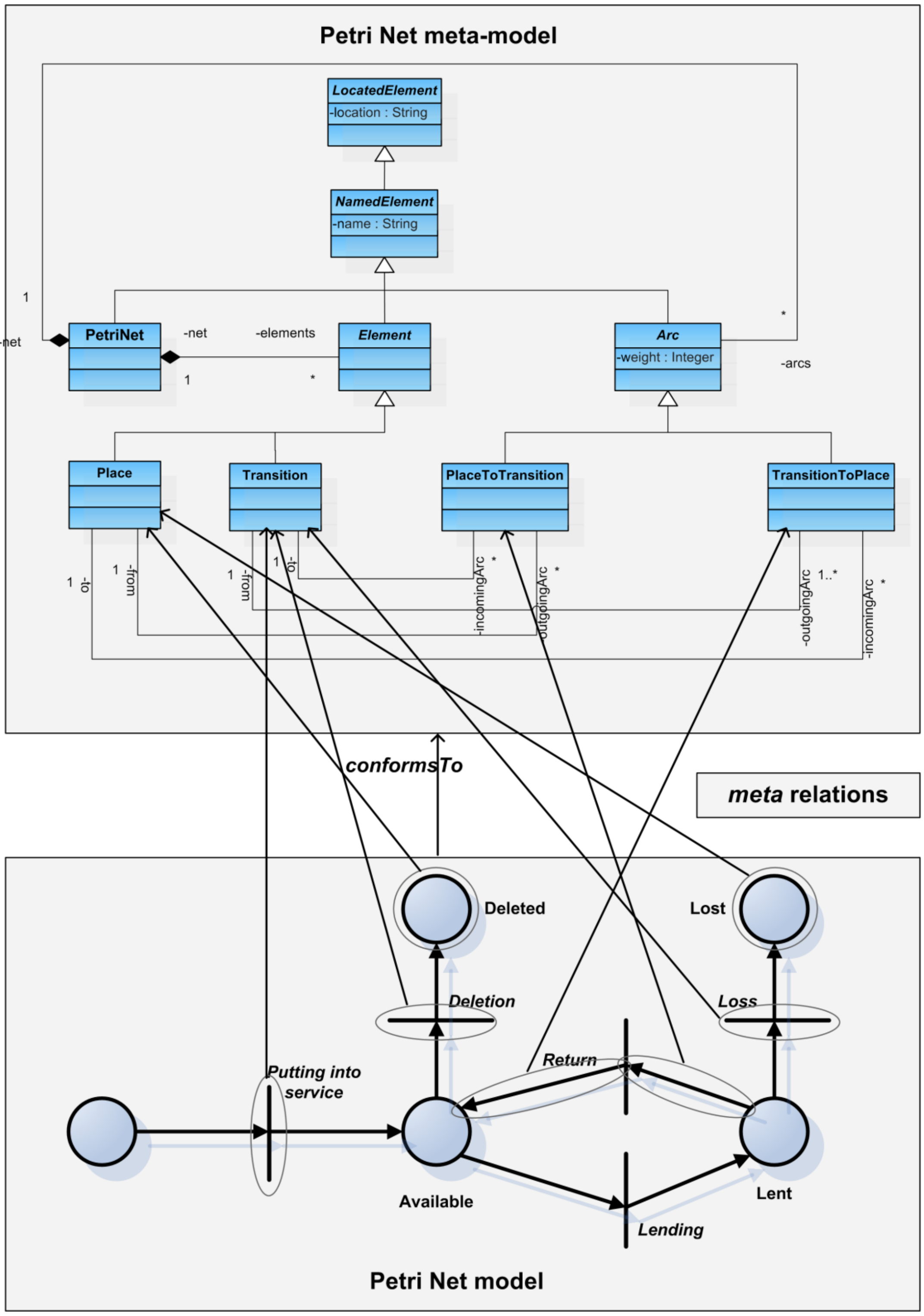


Figure 2.17: Meta relations between Petri Net model and meta-model [ATL, 2006] [Gašević, 2004]

As any other model, a Petri Net model network is composed of a certain number of different elements. In the context of Petri nets, these elements conform to *places*, *transitions* and *arcs*, and they constitute a model. These different elements, together with the way they are connected, conform to the Petri Net meta-model. In the same way, each model conforms to its meta-model. This relation associates each model element with a meta-model element that it instantinates. In addition, the meta-model itself can conform to some meta-metamodel (as it is shown in Figure 2.16, *MLModel*$_{ML}$).

2.2.3. Model transformations

In MDA, model transformations are sequentially applied over models until the system's code is generated. In this section, we describe model transformations and transformation languages in more detail.

2.2.3.1. Definitions and transformation types

The MDA Guide [Miller & Mukerji, 2003] gives a definition of *model transformation*: "*Model transformation is the process of converting one model to another model of the same system*". [Kleppe et al., 2003] defines model transformation as "*automatic generation of the target model from a source model, which conforms to the transformation definition*". Kurtev uses the following definition: "*A model transformation is a process of automatic generation of a target model from a source model, according to a transformation definition, which is expressed in a model transformation language*" [Kurtev, 2005].

Brown defines three types of model transformations [Brown et al., 2005]:

1. *Refactoring transformations* - reorganization of a model based on some well-defined criteria. In this case, the output is a revision of the original model, called the refactored model. An example could be as simple as renaming all the instances where a UML entity name is used, or something more complex like replacing a class with a set of classes and relationships in both the metamodel and in all diagrams displaying those model elements.

2. *Model-to-model transformations* - converting information from one model or models to another model or a set of models, typically where the flow of information is across abstraction boundaries. An example would be the conversion of one type of model into another, such as the transformation of a set of entity classes into a matched set of database schema, Plain Old Java Objects (POJOs), and XML-formatted mapping descriptor files.

3. *Model-to-code transformations* - converting a model element into a code fragment. This is not limited to object-oriented languages such as Java and C++, nor to other programming languages. Configuration, deployment, data definitions, message schemas, and others kinds of files can also be generated from models expressed in notations such as the UML. Model-to-code transformations can be developed for nearly any form of programming language or declarative specification. An example would be to generate Data Definition Language (DDL) code from a logical data model expressed as a UML class diagram.

Detailed transformations classification is given in [Mens & Van Gorp, 2005] and [Czarnecki & Helsen, 2003].

As it can be seen from previous definitions, model transformations give the possibility for target models creation from different number of source models. The goal is to increase productivity and to reduce time of development by using concepts close to the problem domain, rather then to use programming languages [Sendall & Kozaczynski, 2003]. OMG starts from the idea that transformation languages must be defined on the MOF meta-model level, and a consequence is that transformation definitions written in such a language become models (M1 level of the MOF architecture, see section 2.2.1.1. and Figure 2.8), and can be regarded as any other model.

By definition, a transformation is usually capable of transforming a set of source models. There are usually multiple source models. A typical case is when a transformation is designed for model transformations of models written in a specific language. To achieve this, the transformation definition is created based on knowledge about the source and target models. The transformation of a model represented in the source language into target-language representation uses meta-entities defined in the meta-models of that language.

Figure 2.18 shows a process of model transformations and the information needed for such transformations. Model transformations require defining a way for creating a target model that conforms to meta-model B, from a source model that conforms to meta-model A. If meta-model A is equal to meta-model B, then the transformation is *endogenous*; otherwise, it is an *exogenous* transformation [Taentzer, 2006]. Model transformation itself should be also defined as a model. This transformation model should conform to the transformation definition that defines transformation semantics and it represents meta-model, so it must conform to a meta-meta model.

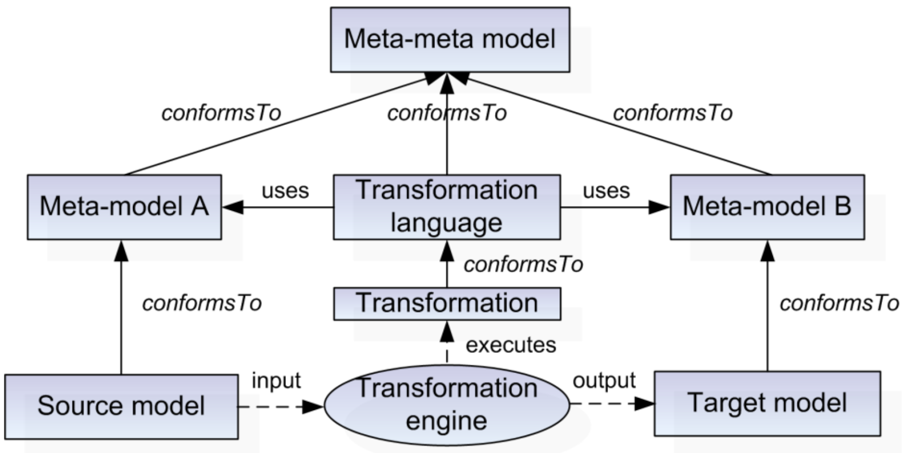


Figure 2.18: Model transformations

2.2.3.2. Model transformation languages

In this section, we describe some classifications of transformation languages based on various criteria defined by [Gardner et al., 2003] and [Czarnecki & Helsen, 2003].

A transformation language is *declarative* if transformation definitions written in that language specify relationships between the elements in the source and target models, without dealing with execution order. Relationships may be specified in terms of functions or inference rules. Transformation engine applies an algorithm over the relationships to produce a result. In contrast, an *imperative* transformation language specifies an explicit sequence of steps to be executed in order to produce the result. An intermediate category may also be introduced, known as *hybrid* transformation languages. These languages have a mix of declarative and imperative constructs.

Some languages allow specification of transformation definitions that can be applied only in one direction: from source to target model. These transformations are known as *unidirectional* transformations. Some other languages (usually declarative ones) allow definitions that may be executed in both directions. These transformations are known as *bidirectional* transformations. This capability is useful when two models must be synchronized. If a transformation language does not support definitions of bidirectional transformations then two separate definitions may be used: one for each direction.

A typical transformation scenario takes one model as input and produces one model as output (*1-to-1* transformation). Generally, there are three other cases: *1-to-N*, *N-to-1* and *M-to-N*. In many cases, multiple models are produced from a single source model (*1-to-N* transformation). For example, a single model may be used to generate Java code and an XML schema used for data exchange. Model composition in which several models are integrated into a single one is an example of *N-to-1* transformation. In the general case, support for *M-to-N* transformations ensures availability of the other three cases.

2.2.3.3. Model transformation tools

It is important to use an appropriate tool that allows representing models and meta-models being transformed by using various formats. Several research groups are creating their own model transformation tools and languages. One of important reasons for choosing a specific language and tools from multiple languages/tools is that the language/tool selected should be the most suitable one for the purposes of the actual transformation work. Today, there are several mature transformation tools and languages. They are mostly available in the form of Eclipse plug-ins and are based on EMF (Eclipse Modeling Framework) for model manipulation. Some of them are commercial, while some of them are not. Here we discuss some of them.

Commercial tools. Borland Together Architect 2006[6] is one of commercial QVT-like transformation tools, which is very close to the OMG's QVT standard (see section 2.2.1.7) and it is heavily based on OCL (see section 2.2.1.6). Its implementation is based on the Eclipse platform. It uses EMF for model representation instead of MOF. Together supports operational transformations and has two types of transformation methods: Mappings and Queries. It gives the possibility to define model-to-model and model-to-text transformations using Together QVT Editor (text-based) or standard Java editor. Input models are in ECore format that can be attained on three ways, from Rose (mdl) models, XDE (mdx) models and XMI for other modeling tools. For model-to-model transformations, Together uses operational QVT and Java, and for model-to-text transformations, it uses Java (with JET). A good support for this tool can be obtained in Borland Developer Network.

IBM has its own Model Transformation Framework (MTF) tool[7]. This framework is a part of IBM's involvement in the QVT standardization, and IBM has developed this prototype model transformation toolkit. It is not QVT-compliant yet. IBM MTF provides functionality to define mappings between meta-models and execution of the model transformations. As input for model files, this tool uses ECore XMI format. The definition of transformation rules is based on relations (text-based) and is expressed in the language called the Relation Definition Language (RDL). IBM MTF is currently in the early phase of the development and is expected to become a more mature tool in the future.

Open source projects. There are also some open source projects for model transformations, like: ATL (ATLAS Transformation Language), UMLAUT NG, YATL (Yet Another Transformation Language), BOTL, FUUT-je, MTL, UMT, Mod-Transf and Tefkat.

UMLAUT NG (Unified Modeling Language All pUrposes Transformer, Next Generation)[8] allows creation of transformations for any model that conforms to some meta-model. It does not depend of the version of UML. UMLAUT transformation framework is based on the tool called UMLAUT NG (next generation) and on a language more appropriate for model transformations: Kermeta. UMLAUT is built on various technologies, including a formal specification based on the OCL at both the model and the meta-model levels, and validation of distributed software systems based on model-checking technologies. UMLAUT transformation rules translate a UML specification as an Abstract Syntax Tree. The core engine of the UMLAUT tool is based on a Transformation Framework from explorations of the Abstract Syntax Tree (AST). It is possible to load a model in XMI, MDL, and UP(Native) file formats. This tool is still in the phase of development and a part of Triskell project.

[6] Borland Together - http://www.borland.com/together.
[7] IBM - Model Transformation Framework, http://www.alphaworks.ibm.com/tech/mtf.
[8] Triskell Project, UMLAUT NG, http://www.irisa.fr/UMLAUT.

The Yet Another Transformation Language (YATL) is a transformation language developed within the Kent Modelling Framework (KMF)[9]. YATL is a hybrid language designed to express model transformations and to answer the QVT Request For Proposals [QVT RFP, 2005]. YATL is developed at the University of Kent. It is described by an abstract syntax (a MOF meta-metamodel) and a textual concrete syntax (BNF). It does not provide yet a graphical concrete syntax as QVT RFP suggested [QVT RFP, 2005]. A transformation model in YATL is expressed as a set of transformation rules. It has a possibility to transform PIMs to PSMs. A YATL transformation is unidirectional. The source and target models are defined using a MOF editor (e.g., Rational Rose or Poseidon) and KMF-Studio is used to generate Java implementations of the source and target models. The source model repository is populated using either Java hand-written code or GUI generated code provided by the modeling tool generated by KMF-Studio. The major part of development of this tool was done before 2005.

The Bidirectional Object-oriented Transformation Language (BOTL)[10] is a graphical language that allows specifying transformations of object-oriented models. It is based upon a precise, formal foundation and a comprehensible graphical notation. Transformation rules are specified using an UML-like notation. Rule and meta-models can be specified with the ArgoUML extension. A verification component (that is not yet realized) verifies whether the rules will produce valid (metamodel-conformant) output or not. BOTL transformation specifications are stored as XML documents and can be consumed by a transformer component that transforms BOTL representations of object models. BOTL uses BOTL rules and XML meta-model as the Transformer. As for the output format, it can generate XML, BOTL file, and Java file. Adaptors can be used to import and export object models from different technical notations, like e.g., Java object structures or XMI representations. Currently, there is an adaptor for Java objects; an XMI adaptor is under development. Since BOTL always transforms object models, the source model is a class model that can be mapped to UML or MOF meta class. This tool has limited complexity and poor performance. Its community is not very developed because this tool is still in the research phase of development.

FUUT-je (Fantastic, Unique, UML Tool for the Java Environment)[11] is a part of the Eclipse GMT (Generative Transformer Project) and it is based on text/XML template rules. It has a possibility to automatically generate a UML model from a well-defined XML document. Then it is possible to generate Java code from UML model. The FUUT-je tool consists of a plugin-loader and a set of plugins that adhere to API and that can communicate via a notification framework. FUUT-je optimizes Java environment. First, an XML schema (a UML class diagram) is imported as a source model. Then it generates application code using the Swing GUI. A main disadvantage of this tool is that it does not allow complex transformations.

MTL (Model Transformation engine)[12] tool is developed by the Triskell team, in Inria. The MTL engine is aimed at helping to resolve the QVT language choice. The Triskell team proposes an architecture using a pivot meta-model. The meta-model language for supporting the pivot meta-model is called either MTL or BasicMTL in the MT engine. The concept of the MT engine is to transfer the source model into the meta-model and then to code directly. The source model can be an XMI using MDR and there is no intermediate model. Loading and saving the model is repository-dependent. For MTL Java is used as the transformer.

UMT (UML Model Transformation)[13] supports model transformation and direct code generation. The tool has its own UML models in the form of XMI (called XMI Light) which is an intermediate format in UMT. UMT transforms XMI to the XMI Light format, which is used by the tool for browsing and editing. It provides also an environment in which new generators can be plugged in.

[9] Yet Another Transformation Language (YATL), http://www.cs.kent.ac.uk/kmf.
[10] The Bidirectional Object oriented Transformation Language (BOTL), http://www4.in.tum.de/~marschal/botl.
[11] Fantastic, Unique, UML Tool for the Java Environment (FUUT-je), http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/fuut/index.html.
[12] Model Transformation Language (MTL), http://modelware.inria.fr/rubrique8.html.
[13] UML Model Transformation Tool (UMT), http://umt-qvt.sourceforge.net.

Two Generators are implemented in XSLT and Java. The formats supported are: GML, WSDL, XML Schema, SQL, Java interfaces, IDL, EJB with Xdoclet tags and BPEL. XMI is used as the source model in UMT. Two transformers are available in UMT; one is implemented in XLST and the other one is in Java. The main usage scenario is to import a UML XMI model and generate code for the desired target platform.

Mod-Transf[14] is an XML- and rule-based transformation language. The rules can be expressed using an XML syntax and can be declarative as well as imperative. There is a possibility to use inheritance and multi-input and multi-output model. The transformation is done by submitting a concept to the engine. It is possible to generate code, also. Mod-Transf supports as input or output models: JMI models, XML, and graphs of objects. It uses MOF as the meta-meta-model.

Tefkat[15] is a declarative, logic-based model transformation language defined in terms of a MOF meta-metamodel. It is implemented as an Eclipse plug-in and uses EMF to handle models based on MOF, UML2, and XML Schema. Tefkat has a concrete SQL-like syntax that is specifically designed for writing scalable transformation specifications using high-level domain concepts rather than operating directly on the XML syntax. For input and output model files, it uses XMI format. It has a good support and tutorials.

In this research it was decided to use ATL [ATL, 2007] as the primary language and tool for model transformations. The decision is based on the following arguments: the biggest user community, a solid developer support, a rich knowledge base of model transformation examples and projects, and a very mature support for various model and meta-model formats. As from January 2007, ATL has been recognized as a standard solution for model transformations in the Eclipse modeling project, and it is now integrated into the new M2M (Model-to-Model[16]) project. ATL is a hybrid transformation language (declarative and imperative), and is based on the OMG OCL (Object Constraint Language) norm for both its data types and its declarative expressions [ATL, 2006]. ATL and QVT share some common features as they initially shared the same set of requirements defined in QVT RFP [QVT RFP, 2005]. However, the actual ATL implementation is different from the QVT standard, although the QVT standard defines requirements primarily for tools, and not for languages [QVT, 2005]. A new version of the ATL compiler (called ATL '2006) supports some advanced functionalities which other tools do not have, such as several source pattern elements in a transformations rule, rule inheritance, invocation of super helpers and *endpoint* called rules (rules that are called last in transformation, after execution of entrypoint, matched and called rules). This tool is very stable and mature. To support this statement, we refer to a very rich documentation that describes the use of this tool. In addition to this documentation, there is a large number of transformations available at the homepage of the ATL tool [ATL, 2007]. The declarative part of ATL is based on *matched* rules notation. Such a rule consists of an input pattern that is connected to source models, and an output pattern that is created in target models. Figure 2.19 shows an ATL rule, with the XML meta-model `Root` element in its input pattern, and with the R2ML meta-model `RuleBase` element in its output pattern.

```
rule RuleBase {
        from    i : XML!Root
        to      o : R2ML!RuleBase (
                      ruleBaseID <- i.getAttrVal('xmlns:r2ml'),
                      rules <- XML!Element.allInstances()->select(e | e.name =
                                                                   'r2ml:DerivationRuleSet' or
                                                                   e.name =
                                                                   'r2ml:IntegrityRuleSet')
                )
}
```

Figure 2.19: An example of ATL matched rule

[14] Mod-Transf, http://www.lifl.fr/west/modtransf.
[15] Tefkat, http://www.dstc.edu.au/tefkat.
[16] Model-to-Model (M2M) project, http://www.eclipse.org/m2m/.

ATL has two imperative constructs: *called* rule and *action* block. A called rule is invoked as a procedure, whereas action blocks are sequences of imperative constructs that can be used in both *matched* and *called* rules. An example of a called rule and an action block is shown in Figure 2.20. The rule in this figure defines the input point for transformation, creates a `Metamodel` element, and assigns it to the *metamodel* variable, which is located in the actual transformation (module).

```
entrypoint rule Metamodel() {
        to
                t : KM3!Metamodel
        do {
                thisModule.metamodel <- t;
        }
}
```

Figure 2.20: A called rule and an action block in ATL

Table 2.2. summarizes transformation tools features, and Table 2.3 (below) shows which tools support some MOF 2.0 QVT features (Y - Yes, N - No, P - Partially) [QVT RFP, 2005].

Table 2.2: Summary of transformation tools features

Tool	**License**	**Input/Output formats**	**Strengths**	**Weaknesses**
Borland Together Architect 2006	Commercial	*Input:* Rose (mdl) models, XDE (mdx) models and XMI. *Output:* Code and XMI.	Supports operational transformations. Fully supports OCL 2.0 expressions. Implementation based on the MOF 2.0 QVT RFP. Reusable libraries of QVT mappings.	Not 100% automated, customizations might be required.
IBM MTF	Commercial	*Input:* XMI, Java Code, XSD. *Output:* XMI, XSD, HTML.	It includes incremental, multi-directional, generic and extensible transformations, traceability, updates and consistency checks.	Not QVT compliant. Low performance.
ATL	Open source	*Input:* XMI. *Output:* XMI.	Complete transformation model from code generation, generic re-usable rules, support for complex transformations, several source pattern elements, rule inheritance, super helpers, good support and wide community.	M-N transformation is not supported by ATL yet. Slow compilation of large-scale transformations.
UMLAUT NG	Open source	*Input:* CDIF, UP and Rose (mdl) models. *Output:* Code.	Simulator, code generation.	It is in early phase od development and does not have good support yet.
YATL	Open source	*Input:* XMI. *Output:* XMI, Java code.	Declarative and imperative style, namespaces, efficient transformations.	Not complaint to the QVT standard. Does not provide yet a graphical concrete syntax.

BOTL	Open source	*Input:* XMI, XML and Botl. *Output:* Botl, XML, and Java code.	Complete generation of target, declarative, bi-directional, graphical, rule-based principle.	Focus only on the transformation of class diagrams. XMI input is under construction.
FUUT-je	Open source	*Input:* XSD, GME, Java, XML, XMI. *Output:* Java code, XML.	Prototype transformer, Java GUI code generator, good performance.	It is still in a very early phase of development.
MTL	Open source	*Input:* MTL, Java, XMI. *Output:* XMI.	Meta-model language (PIM model) for model transformation, but it does not adress implementation.	Tool set for supporting QVT features are not jet available (with the exception of the compiler for meta-model transformation language).
UMT	Open source	*Input:* XMI (light). *Output:* XMI, IDL, XDoclet, PBEL, Java, SQL, XSD, Workflow, WSDL.	It has a simple implementation because it uses MDA "light" approach, in which no PSM model is used. Documetation is comprehensive.	Query and View are supported only theoretically by XPath.
Mod-Transf	Open source	*Input:* XMI, XSD, graph of objects (XMI). *Output:* XMI, Code.	Declarative and imperative style, inheritance, multi-intput and output model, customizable.	Text-only rules, doesn't have a GUI editor, poor performance.
Tefkat	Open source	*Input:* XMI. *Output:* XMI.	Transformations are constructive, using constraints, model-merge.	Rule cannot depend on its own negation. Queries are limited only to source extents.

Table 2.3: Characteristics of analysed transformation tools

Tool	**Self-containinment**	**Scalability**	**Simplicity**	**Bi-directional mappings**	**Ease of Adoption**	**Rich conditions**	**Abstract syntax**	**Adoption of common terminology**	**Support for composition and reuse**	**Complex transformations**	**Examples provided**	**Robust on transformation executions**	**Tooling aspect**
Together	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y
MTF	Y	N	Y	Y	Y	Y	N	Y	Y	N	N	N	N
ATL	Y	N	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y
UMLAUT-NG	Y	N	Y	N	Y	N	N	N	N	N	N	N	Y
YATL	Y	N	Y	Y	Y	N	N	Y	Y	Y	Y	N	Y
BOTL	Y	N	Y	N	Y	N	N	Y	N	Y	Y	N	N
FUUT-je	Y	Y	Y	N	Y	Y	Y	Y	Y	N	N	N	N

MTL	Y	Y	N	N	Y	Y	Y	Y	N	N	Y	N	N
UMT	Y	Y	Y	Y	Y	Y	N	Y	N	N	Y	N	N
Mod-Transf	Y	N	Y	N	Y	N	N	Y	N	N	Y	N	Y
Tefkat	Y	N	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y

In addition to the tools analyzed in this section, there are some other transformation tools in the early stage of development, such as: ModelMorf[17], GReAT[18], SmartQVT[19] and VIATRA 2[20]. More details about them can be found in their Web pages.

2.2.3.4. Model transformation types

In practice, model transformations can be applied in several ways. In MDA, there are four categories of techniques for applying model transformations [Brown et al., 2005]:

- *Manual* - The developer examines the input model and manually creates or edits the elements in the transformed model. The developer interprets the information in the model and makes modifications accordingly.
- *Prepared Profile* - A profile is an extension of the UML semantics in which a model type is derived. Applying a profile defines rules by which a model is transformed.
- *Patterns* - A pattern is a particular arrangement of model elements. Patterns can be applied to a model and result in the creation of new model elements in the transformed model.
- *Automatic* - Automatic transformations apply a set of changes to one or more models based on predefined transformation rules. These rules may be implicit to the tools being used, or may have been explicitly defined based on domain specific knowledge. This type of transformation requires the input model to be sufficiently complete both syntactically and semantically, and may require models to be marked with information specific to the transformations being applied.

The use of profiles and patterns usually involves developer's input at the time of transformation, or requires the input model to be “marked”. A marked model contains extra information not necessarily relevant to the model’s viewpoint or level of abstraction. This information is only relevant to the tools or processes that transform the model.

2.3. ECLIPSE MODELING FRAMEWORK (EMF)

Eclipse Modeling Framework (EMF) is a conceptual modeling framework for Eclipse [Budinsky et al. 2003]. Eclipse is an open-source project lead by a consortium of companies, IBM being among them, with the goal to provide a highly integrative tool platform. Its current version is 2.2.1 (January 2007.) Eclipse includes a core and generic environment for tool integration and a Java environment for development that is built by using that core. Other projects use the basic core to support different types of tools and development environments. The projects in Eclipse are implemented in Java and can be run on most operating systems.

The Central part of the EMF-based modeling is a model, which includes a set of elements defined by UML and its standard notation. It is a UML class diagram in the first place. In the EMF, a model is not that general and high-level as it is usually assumed.

The EMF does not require a complete, distinct methodology or some sophisticated tools for modeling. Eclipse Java Development tools are the only tools that are really needed. EMF connects

[17] http://www.tcs-trddc.com/modelmorf/index.htm.
[18] http://www.isis.vanderbilt.edu/projects/gme/.
[19] http://smartqvt.elibel.tm.fr/.
[20]http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html.

modeling concepts directly with their implementations, thus bringing Eclipse and Java programmers closer, which results in modeling possibilities that are easy to learn.

2.3.1. Basic concepts of the Eclipse Modeling Framework

EMF is a Java-based environment for development of tools and other applications based on a structured model. It enables to develop a complete model for an application by using UML diagrams. This model can be used only for documentation, or it can be used as input for generating a part of an application or the complete application. This class of modeling usually requires expensive tools for object-oriented analysis and design. EMF is often used as a model handler, by model transformation tools (bin addition to NetBeans MDR). An important characteristic of the EMF is that it offers a "low entry price" because it requires only a small portion of UML modeling (classes and their attributes and relations), i.e. only a graphical modeling tool. EMF uses XMI for storing model definitions. To create such a document, there are four options:

1. creation of an XMI document, directly, by using an XML or text editor;
2. XMI document export from modeling tools (such as IBM Rational Rose);
3. annotation of Java interfaces with model attributes;
4. using XML Schema to describe the form of model serialization.

The first and the third approach require knowledge of XML and Java, respectively, which is good if the author is familiar with these technologies. The second approach is preferred if we use a modeling tool. The last approach is suitable for creating applications that must read or write some XML content to a file.

EMF consists of three fundamental pieces: Core, EMF.Edit and EMF.Codegen. Core provides a basic support for generating and executing classes implemented in Java for a model. It includes a meta model (ECore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and an efficient reflective API for manipulating EMF objects generically. EMF.Edit includes generic reusable classes for building editors for EMF models and extends the Core by adding support for generating adapter classes that enable preview and work with the model, as well as a basic (visual) editor for the model. It also has a command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo. EMF code generation facility (EMF.Codegen) is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

An overview of possibilities and the process of creating an ECore model are shown in Figure 2.21.

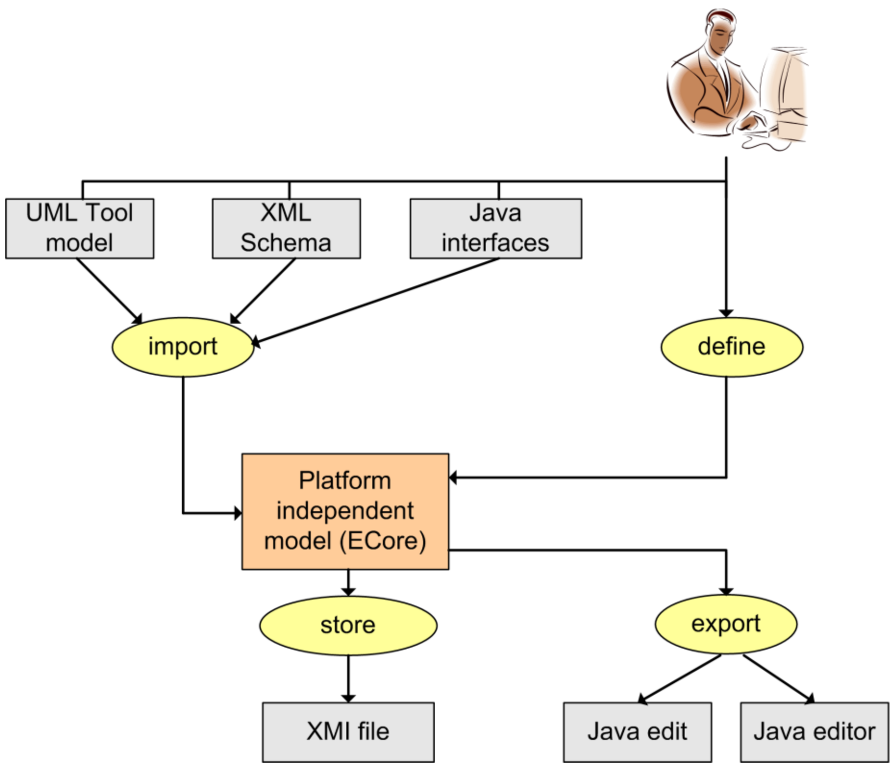


Figure 2.21: Creating a platform independent ECore model

EMF also supports three levels of code generation (from the model):

- *Model* - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta-data) implementation classes;
- *Adapters* - generates implementation classes (called *ItemProviders*) that adapt the model classes for editing and display;
- *Editor* - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

2.3.2. Working with the Eclipse Modeling Framework

The work with EMF is illustrated here using the example of a simple library meta-model (an excerpt from [EMF2005]). The meta-model of the library is defined in a UML tool (IBM Rational Rose) and is shown in Figure 2.22.

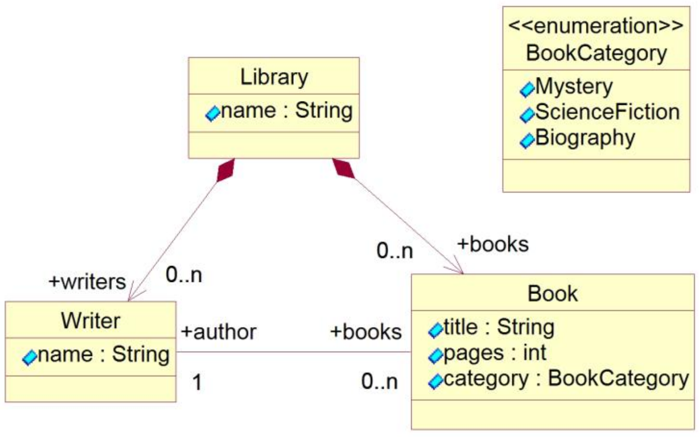
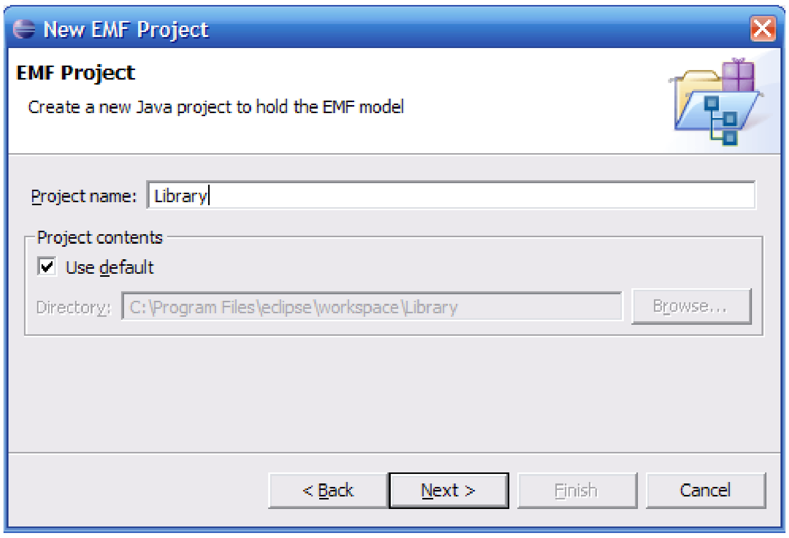


Figure 2.22: Library meta-model

We start by describing the process of creating an EMF model and generating a simple model editor for it. To import a meta-model created by IBM Rational Rose from a file, it is needed to take the following steps in the Eclipse environment (which must have EMF installed[21]): File/New/Project... dialog, then expand "Eclipse Modeling Framework" and select "EMF Project". Click the "Next" button. In the dialog box that appears, it is necessary to give a name to the project (in this case it will be "Library", Figure 2.23).


New EMF Project

EMF Project

Create a new Java project to hold the EMF model

Project name: Library

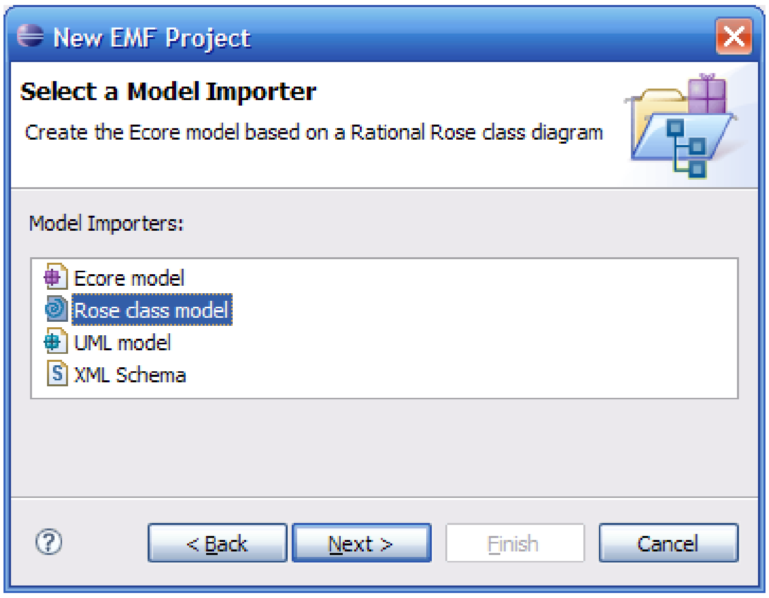
Project contents

☑ Use default

Directory: C:\Program Files\eclipse\workspace\Library

Figure 2.23: Creating a new EMF project

After entering the project name, it is necessary to select a Model Importer (which can be ECore model, Rose class model, and XML Schema, or even UML2 model if UML2 Eclipse plug-in[22] is installed[23]), and the location on the drive where the model file is located (Figure 2.24).


New EMF Project

Select a Model Importer

Create the Ecore model based on a Rational Rose class diagram

Model Importers:

Ecore model
Rose class model
UML model
XML Schema

Figure 2.24: Model Importer

[21] http://www.eclipse.org/modeling/emf/downloads/.
[22] http://www.eclipse.org/modeling/mdt/?project=uml2.
[23] If UML2 plug-in is used as Model Importer, in that way a UML model which is exported from some UML tool, which have support for exporting in EMF UML2 XMI format, can be translated to ECore model.

After choosing the file, the EMF "wizard" will offer to create a .ecore file package in the input file. In our case, this package will be the "library" from which the meta-model file (library.ecore) and the generator meta-model (library.genmodel) will be created. The latter, which controls code generation for the meta-model, is opened in the main view (Figure 2.25).

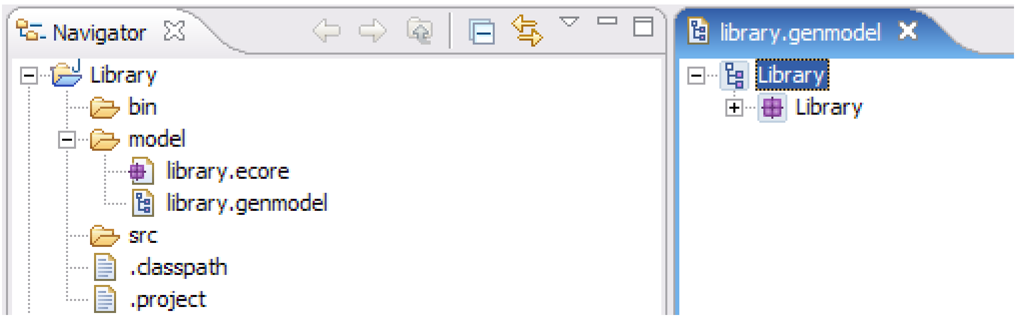


Figure 2.25: Project and files created for the imported meta-model

The model generator shows a root object that represents the meta-model as a whole. This object has "children" that represent packages, and their "children" represent classifiers (classes and data types). "Children" classes are attributes, references and operators (Figure 2.26).

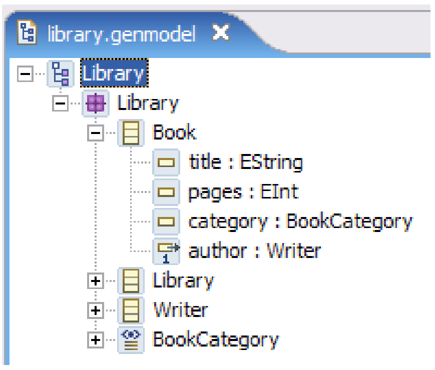
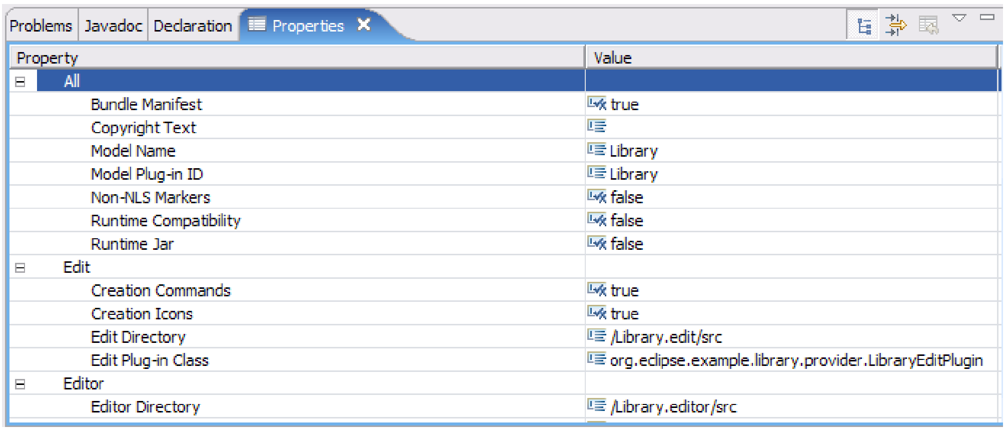


Figure 2.26: Generated meta-model elements

Each generated element has properties, which can be viewed in the Properties window. This window can be shown if we right-click the "Library" model and then choose "Show Properties View" from the pop-up menu (Figure 2.27). In the Properties window, it is possible to change properties of all elements in the meta-model.



Property	Value
⊟ All	
Bundle Manifest	true
Copyright Text	
Model Name	Library
Model Plug-in ID	Library
Non-NLS Markers	false
Runtime Compatibility	false
Runtime Jar	false
⊟ Edit	
Creation Commands	true
Creation Icons	true
Edit Directory	/Library.edit/src
Edit Plug-in Class	org.eclipse.example.library.provider.LibraryEditPlugin
⊟ Editor	
Editor Directory	/Library.editor/src

Figure 2.27: Properties window – meta-model element attributes

Model generator is the place where code generation for the meta-model is initiated. This can be done by right-clicking the "Library" object (the root object of the GenModel) and selecting "Generate Model Code" from the pop-up menu. After the Java code is generated, a new pair of Java interfaces will be created for the package and the model elements factory (Figure 2.28). In this window, we can see two new packages, with "impl" and "util" sufixes, which contain implementations of these interfaces and additional classes. When the model is modified, Java interfaces and classes can be generated for it again (in the same way).

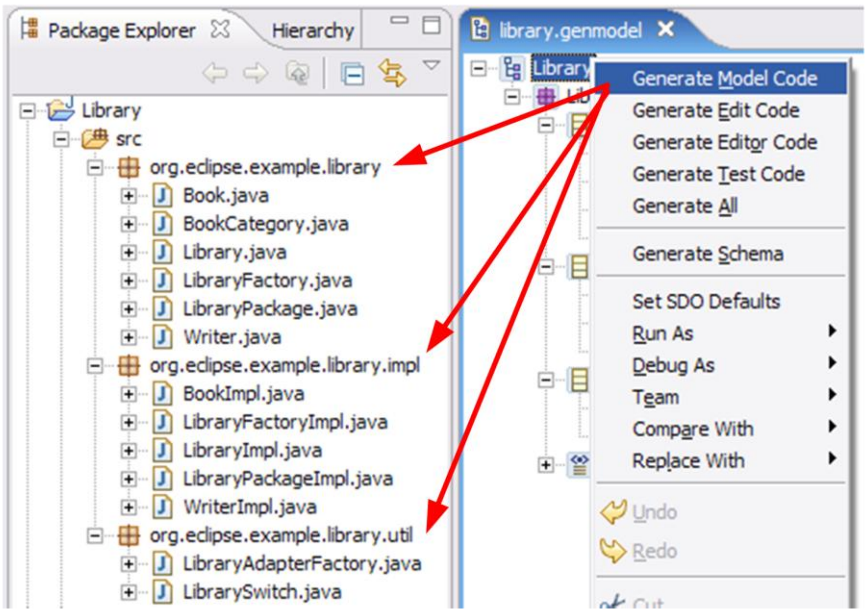


Figure 2.28: Generated Java interfaces and implementation classes for the meta-model

In addition to the code, a fully functional Eclipse editor can also be generated for any model. By default, it is split between two plug-ins. An "edit" plug-in includes adapters that provide a structured view and perform command-based editing of the meta-model objects. An "editor" plug-in provides the UI for the editor and a wizard. Also, it is possible to create an XML Schema for any meta-model. In our example, i.e., in the model generator (.genmodel file) that appears as a pop-up menu on right-clicking the "Library" meta-model object, the "editor" and "edit" plug-ins are created by choosing "Generate Edit Code" and "Generator Editor Code". In the Package Explorer, we can see two new packages, with "edit" and "editor" suffixes (Figure 2.29).

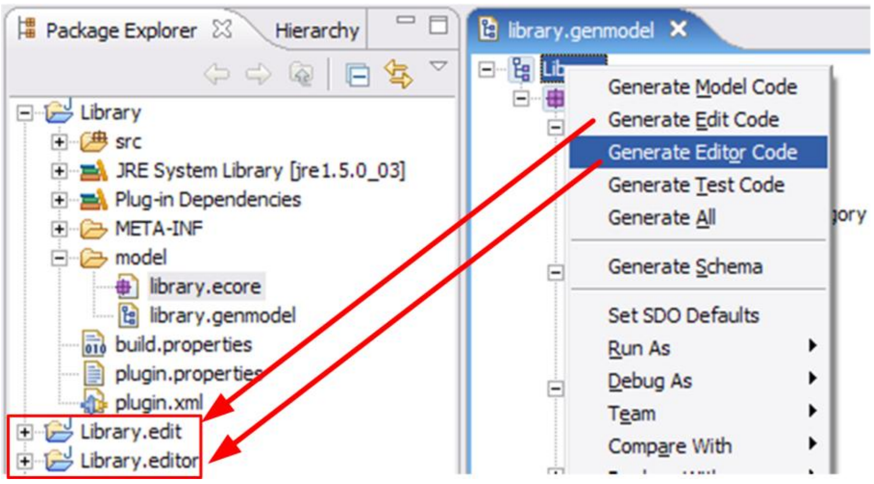


Figure 2.29: Generated packages for the "edit" and "editor" plug-ins

To run the generated editor, we need to run another Eclipse instance (Run/Run As/Eclipse Application). These plug-ins will be run in this instance. The Library model wizard can be used to run the new meta-model instance. An empty project is created first (File/New/Project, then Simple and Project). In the dialog that appears, the name of the project should be inserted (e.g. "Library project").

Now we can create a meta-model instance, by selecting "New/Other..." from the pop-up menu that appears when right-clicking on the created project. In the resulting dialog (for Wizards), we choose "Example EMF Model Creation Wizards" and "Library model", and then click "Next". In the next dialog (Library model), a filename should be specified (with the extension .library). The created library model is opened in the main window (with Library Model Editor). The root object conforms to the file in which it is created. When "platform:/resource/Library project/My.library" is expanded, the Library object can be seen (Figure 2.30).

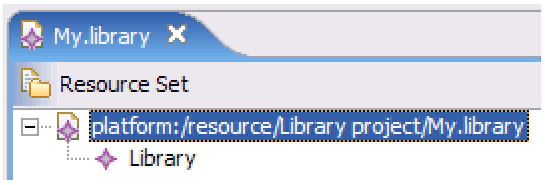
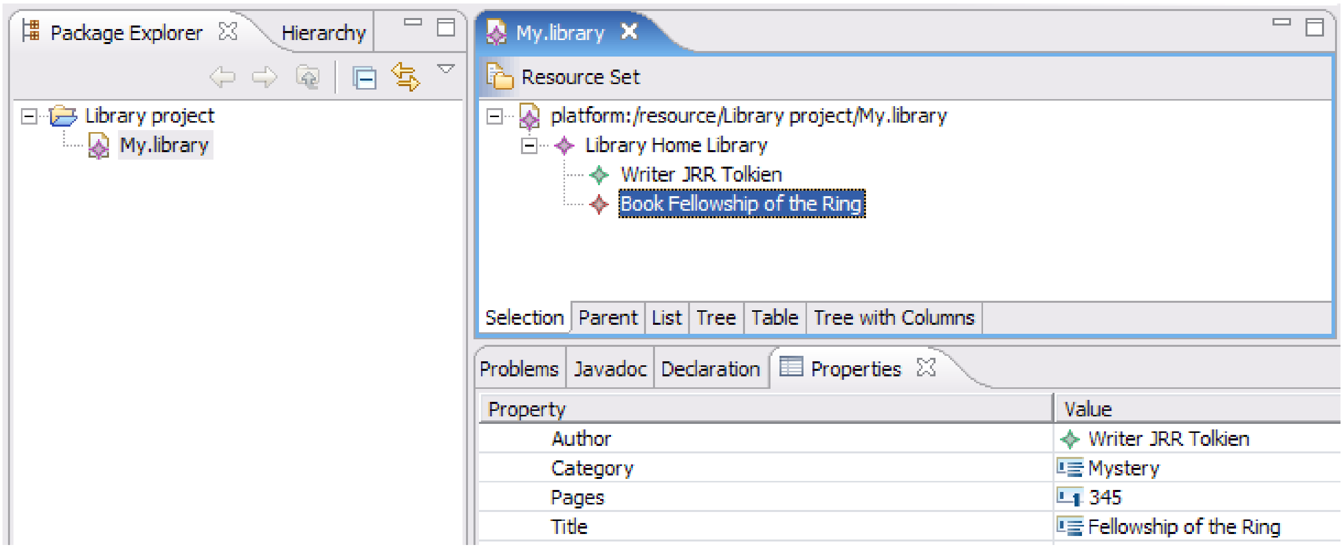


Figure 2.30: Library object

In the Properties window, for the "Library" item we can add a name (the "name" property). This change will be shown in the editor too. To create a certain book or a writer, we need to right-click the "Library" item in the editor, and choose "New Child", and then "Writer" or "Book". For the generated element, we can enter its specific attributes in the Properties window. All changes to attributes are shown in the editor (Figure 2.31). The concrete library model reflected in this example is shown in Figure 2.32.



Property	Value
Author	Writer JRR Tolkien
Category	Mystery
Pages	345
Title	Fellowship of the Ring

Figure 2.31: Concrete library model in the generated Library Model Editor

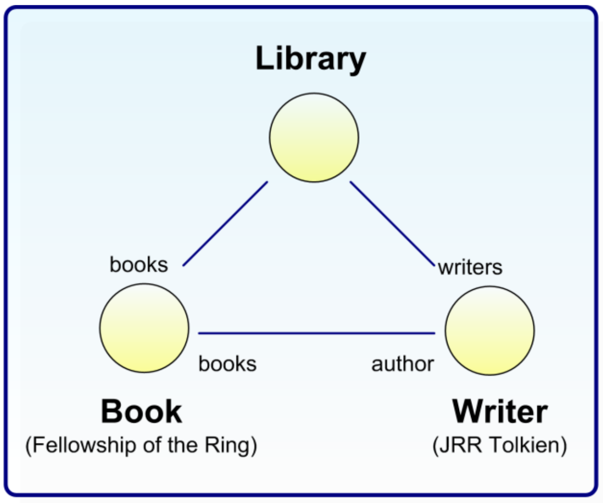
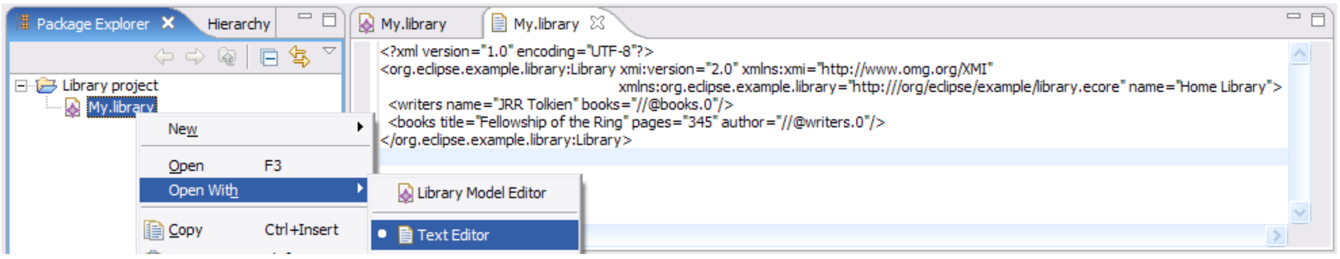


Figure 2.32: Conceptual model of the concrete library

The library model shown above can be opened by using a standard text editor in Eclipse: right-click My.library, choose "Open With..." and "Text Editor". The main window will show the content of the .ecore file (in the EMF XMI 2.0 format) that contains the model (Figure 2.33).



```
<?xml version="1.0" encoding="UTF-8"?>
<org.eclipse.example.library:Library xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                                     xmlns:org.eclipse.example.library="http:///org/eclipse/example/library.ecore" name="Home Library">
  <writers name="JRR Tolkien" books="//@books.0"/>
  <books title="Fellowship of the Ring" pages="345" author="//@writers.0"/>
</org.eclipse.example.library:Library>
```

Figure 2.33: Contents of the concrete library model file (.ecore) in a text editor

To increase productivity, construction of applications using EMF offers multiple possibilities: notification about changes, support for presentation including basic XMI- and Schema-based XML serialization, and software environment for model validation and a very efficient reflexive API for EMF object manipulation. Most importantly, EMF supports interoperability with other EMF-based tools and applications. EMF is still a very comprehensive and complex environment for modeling. It also has a reflective Dynamic EMF API that provides a dynamic implementation of the reflective API (that is, the *EObject* interface) which, although slower than the one provided by the generated classes, implements exactly the same behavior. This API can be used for EMF-based model manipulation in any Java application.

It is important to emphasize the relation between EMF (i.e. ECore) and MOF. EMF has started as an implementation of the MOF specification, but has then evolved from it. This was based on the experience from the implementation of a large number of tools that use EMF. In fact, EMF can bee seen as a highly efficient implementation of MOF API core. In EMF, the meta-model of its core is called ECore (similarly to EMOF in MOF version 2.0). Nevertheless, there is a small number of differences between ECore and EMOF, mainly in names. However, EMF can transparently read and write EMOF serialization.

2.3.3. ECore modeling concepts

The model used to represent EMF models is called ECore. ECore is itself an EMF model, so we can say that it is the meta-model to itself and is usually used to specify platform independent models. It is actually also a meta-metamodel. There is often a misunderstanding about meta-metamodels, but this concept is actually very simple. Meta-metamodel is just a model of another model, and if that other model is a meta-model to itself, then meta-model is actually meta-metamodel (this concept can recursivelly go to meta-meta-metamodels, but ECore puts a limit here, because it is described by itself). Figure 2.34 shows the ECore model with its core elements (attributes, relations and operations).

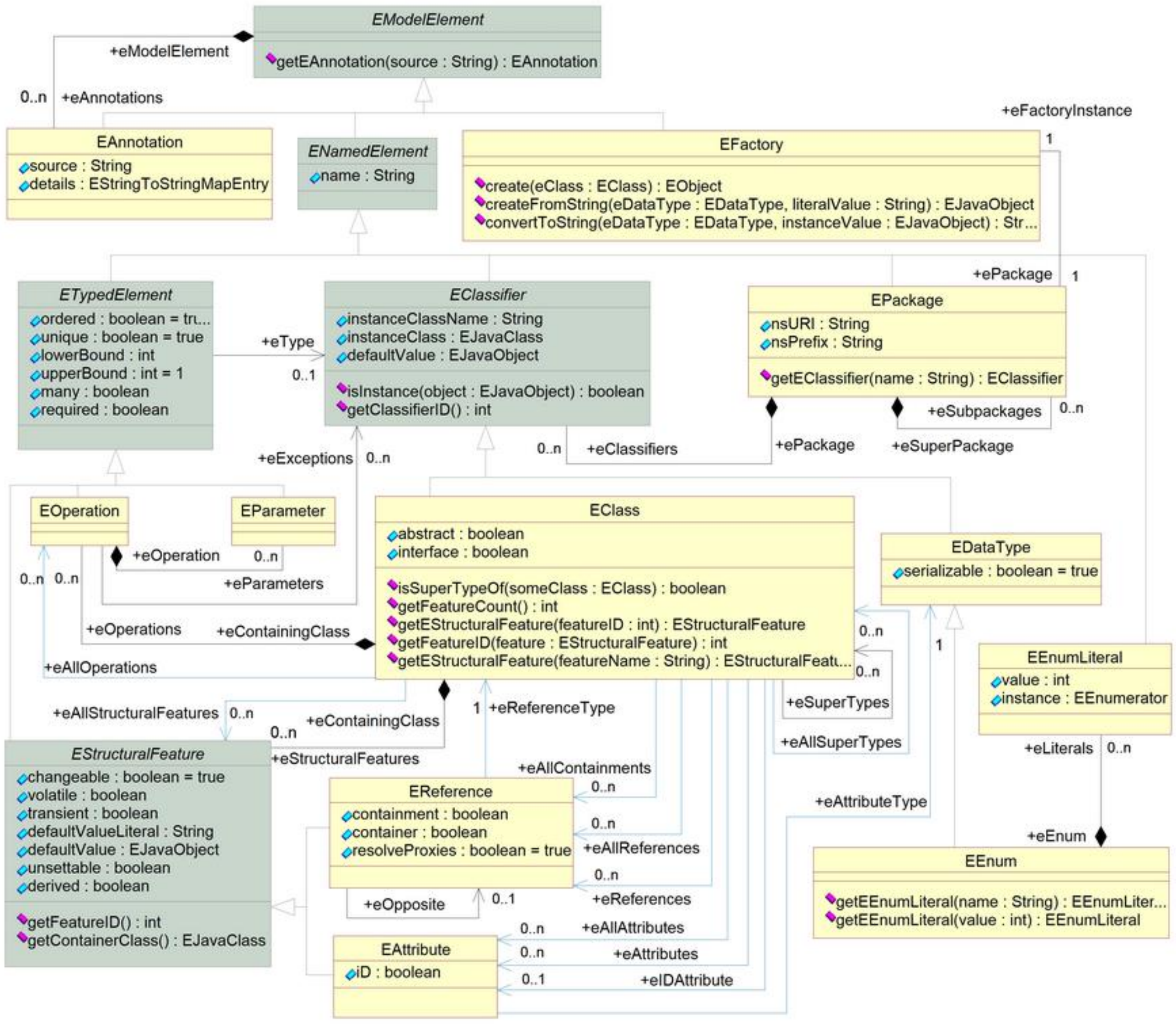


Figure 2.34: ECore model core elements[24]

From the above diagram, we can see that there are four basic ECore classes that are necessary to representat a model:

1. `EClass` - used for representing a modeled class. It has a name, zero or more attributes, and zero or more references.
2. `EAttribute` - used for representing a modeled attribute. Attributes have a name and a type.
3. `EReference` - used for representing an association end between classes. It has a name, a boolean attribute that indicates if it implies containment, or a destination reference type, which is another class.

[24] http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/plugins/org.eclipse.emf.ecore/src/model/Ecore.mdl?revision=1.2

4. `EDataType` - used for representing attribute types. This type can be primitive, such as `int` or `float`, or object type, such as `java.util.Date`.

Here we can see that ECore is a small and simple subset of the UML. Full UML supports a more sophisticated modeling than it is the case with EMF core (e.g., UML supports much more complex specification of behavior).

The class instances defined in ECore are used for describing the application model class structure. When the classes defined in ECore are extended (inherited) for defining a specific application model, it is called the basic (or core) model. Figure 2.35 shows an example UML class named `Student`, with two `String` attributes (name and surname). EMF generates a corresponding ECore class (such as `EClass`) and represents it with a Java interface and an appropriate implementation class. The `EClass` for the `Student` class is mapped to a Java interface:

```
public interface Student ...
```

and to an appropriate implementation class:

```
public class StudentImpl extends ... implements Student {...}
```

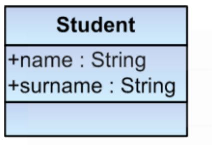


Figure 2.35: UML class - `Student`

This separation of interfaces and implementation is a choice that is enforced in the EMF design. The reason for this separation is to stay in line with good programming practices. It is important to notice that the generated interface directly inherits the `EObject` interface:

```
public interface Student extends EObject { ... }
```

`EObject` is an equivalent to `java.lang.Object` class, which represents the root class for all modeled objects. By inheriting from `EObject`, the following behaviours are inherited:

- *eClass()* - returns the object's meta-object (`EClass`);
- *eContainer()* and *eResource()* return the object's container and resource;
- *eGet()*, *eSet()* and *eUnset()* provide an API for reflexive access to objects.

For each attribute in an interface, two appropriate set/get method signatures are created:

```
String getName();
void setName(String value);
```

Using its notification system (`EObject` interface inherits `Notification` interface), EMF enables simple implementation of relations (references) between classes, i.e., objects. In addition, ECore has the possibility to work with methods through its behaviour attributes (`EOperation` and `EParameter` classes, which represent methods and parameters, respectively), as well as with packages (and element factories), data types and enumeration types.

The ECore class hierarchy is shown in Figure 2.36.

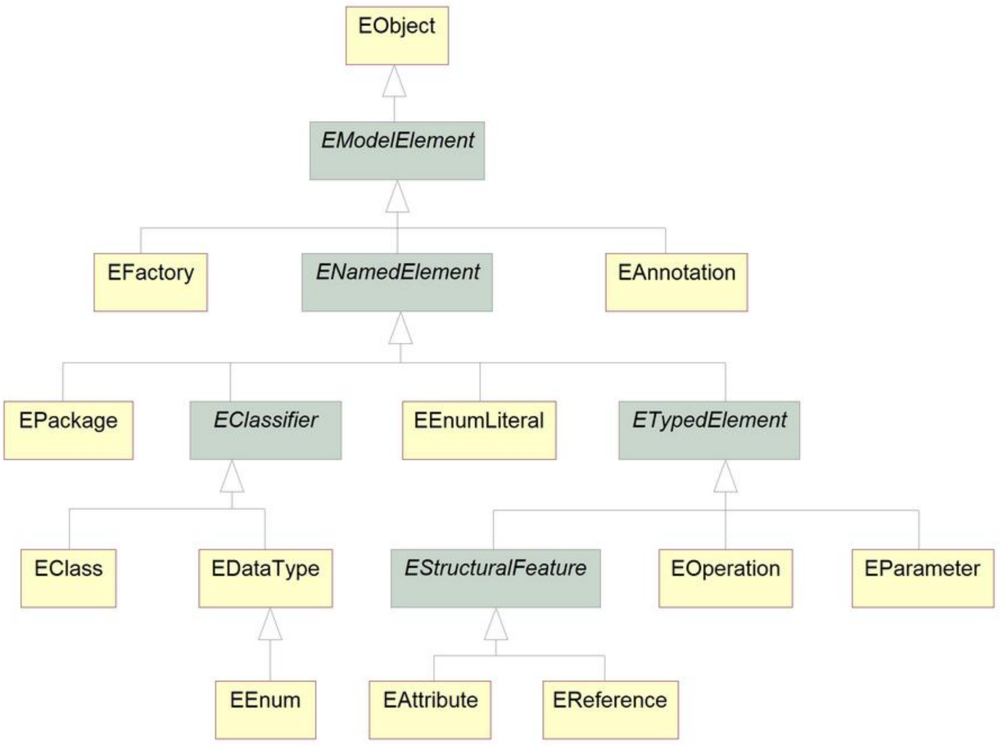


Figure 2.36: ECore class hierarchy

Each ECore model is stored in an XMI file and starts with the definition of a package that contains all other elements (classes, attributes, etc.). An example of an EMF XMI file is shown in Figure 2.37.

```
<ecore:EPackage
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="package">
</ecore:EPackage>
```

Figure 2.37: EMF XMI file that contains one (*ECore*) package

`EPackage` represents a package definition, where *xmi:version* XMI version (OMG), *xmlns:xmi* and *xmlns:ecore* define namespaces for two XML Schemas that are used, and *name* is the package name.

A class (for example `Student`) is defined by the `eClassifiers` tag and the meta-reference *xsi:type="ecore:EClass"*. Attributes are represented as `eStructuralFeatures` with `EAttribute`. An example of a class defined with two attributes in an XMI file is shown in Figure 2.38.

```
<eClassifiers xsi:type="ecore:EClass" name="Student">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="name" lowerBound="1"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/#//EString"/>
      <eStructuralFeatures xsi:type="ecore:EAttribute" name="surname"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/#//EString"/>
</eClassifiers>
```

Figure 2.38: XMI file with the ECore class serialized from the UML class `Student` (Figure 2.35)

2.4. Modeling and technological spaces

A *modeling space* is defined in [Djurić, 2006], as "*a modeling architecture defined by a particular meta-metamodel*". The purpose of this approach is a better understanding of various things that can be modeled, as well as using and combining principles of various software modeling technologies. Meta-models defined by the meta-metamodel and models defined by these meta-models represent the real world from one point of view, i.e., from the point of view of that modeling space. Since the meta-metamodel defines the core concepts used in defining all other metamodeling concepts, it is defined by itself. If it was defined by some other concepts, it would not be a meta-metamodel; it would be an ordinary metamodel in some other modeling space.

Successful modeling must consider a domain in which the modeling will take place. These spaces are formally considered and discussed in [Unhelkar & Henderson-Sellers, 2004].

Figure 2.39 shows a few examples of well-known modeling spaces. All layers above the M0 layer conform to their corresponding upper layers, all the way up to the uppermost level which contains the super-metamodel. It follows from Figure 2.39 that a UML model is in the MOF modeling space. The UML meta-model is defined in the MOF modeling space (which defines itself, because it is reflexive), while each UML model is defined by the UML meta-model. Models at the M1 level are an ECore model, an XML document, and also a Java code. Modeling things from the real world in a modeling space is done by using abstractions from that modeling space. If we model these things in another modeling space, they would be described with different models.

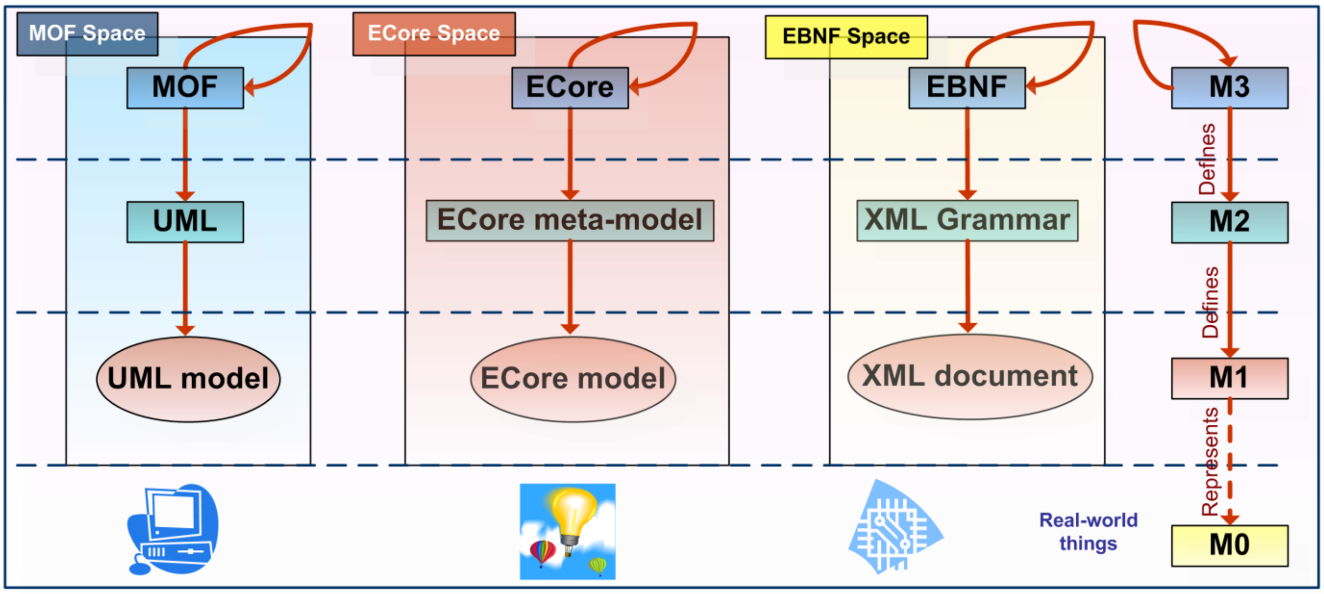


Figure 2.39: MOF, ECore and EBNF modeling spaces

There are two types for modeling spaces: *conceptual* and *concrete*. Conceptual modeling spaces are focused on conceptual (abstract or semantic) things, like models, ontologies, mathematical logics, etc. They are not interested in techniques for representing and sharing their abstractions. However, we must have some techniques to materialize (or serialize) those modeling spaces. We can do this by using *concrete modeling spaces*, which are equipped with notation. Examples of those materializations are some syntax or databases.

If we take the MOF space as an example of a conceptual modeling space, the basic concepts of the MOF meta-metamodel – such as `Class`, `Association`, `Attribute`, `Package` and the relations among them – are expressed using these concepts themselves. We can draw them using UML diagrams, but a group of boxes and lines is not a MOF model – it is a drawing of a MOF model. We can serialize them into XMI, enabling computers to process them and programs to share them – but then we leave the MOF modeling space and enter the EBNF modeling space. One can argue that these drawings, i.e., UML diagrams, or models serialized into XMI, are inside the MOF modeling space because they represent MOF concepts. Indeed, they do represent concepts from the MOF space. Simultaneously, they represent other things from the real world. It means that the MOF concepts are modeled in another modeling space. An example of a concrete modeling space is EBNF. Theoretically well founded, it arguably has some "semantics" primarily for type checking, and has a syntax that is formally specified using a grammar, but it lacks semantics. If we parse the expression a = "Student", we get a syntax tree that does not know that it deals with the name of a student. We always need some external interpretation of what its abstract syntax means. Actually, this interpretation is given in the corresponding models from other technical spaces that were serialized into the EBNF form. Being able to represent bare syntax, concrete modeling spaces need some means to express the semantics, i.e. the meaning of the data they carry. Conceptual modeling spaces, on the other hand, are able to represent semantics, but need a means to represent their information physically. It is obvious that they should complement each other's representation abilities to create models that have both semantics and a syntax.

In addition to categorising modeling spaces in conceptual and concrete spaces, a categorization based on two types of usage scenarios for different modeling spaces is also possible [Djurić et al., 2006]:

- *Paralell spaces* - one modeling space models the same set of real-world things as another modeling space, but in another way. In this case, the relation between these modeling spaces is oriented towards pure transformation, bridging from one space to another. Examples of such parallel modeling spaces are MOF and ECore (ECore is a meta-metamodel very similar to but also different from MOF; although ECore is a little simpler than MOF, they are both based on similar object-oriented concepts). Another example of parallel spaces are a database and an OWL ontology that model a student.
- *Orthogonal spaces* - a modeling space models concepts from another modeling space, taking them as real-world things, i.e., one modeling space is represented in another modeling space. This relation is often used in round-trip engineering to facilitate different stages of modeling some system. For example, in order to make a Java program, we could first use a UML model to create classes and method bodies, then transform this UML model into an ECore model, then generate Java code from this model, and complete the Java program using a Java IDE. Orthogonal modeling spaces are also used when a conceptual modeling space is implemented using a certain concrete modeling space – for example, when one develops an EMF-based repository to run on a Java platform.

A detailed discussion on modeling spaces can be found in [Djurić, 2006] and [Djurić et al., 2006].

Another important concept is that of *technological spaces*, which are defined as: "*working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities*" [Kurtev, 2002]. If we have in mind the definition of modeling spaces, we can say that a technological space is a working context which connects different modeling spaces. It is often associated to a given user community with shared know-how, educational support, common literature and even regular workshop and conference meetings. Examples of some technological spaces are concrete and abstract syntax of programming languages, ontology engineering, XML-based languages, Data Base Management Systems (DBMS), Model-Driven Architecture (MDA) as defined by the OMG., etc. Each space is defined according to a couple of basic concepts: Program/Grammar for the Syntax TS, Document/Schema for the XML TS, Model/Meta-Model for the MDA TS, Ontology/Top-Level Ontology for the Ontology engineering TS and Data/Schema for the DBMS TS. There are bridges between various spaces and these bridges also have specific properties. Some are bi-directional, and others are uni-directional (one-way) bridges. Some operations may be performed easier in one space and the result may then be imported into another space. For such an operation, software engineers need to compare the cost of achieving it in two or more spaces and also to evaluate the costs of export/import between the spaces. Abstract syntax technological space is the space with the longest history rooted at the days when computer engineers started building abstractions over the native machine language and expressing these abstractions with languages closer to the problem domain. It is based on solid theoretical foundations: context-free grammars for specification of language syntax, a number of formalisms for specification of language semantics (attribute grammars, denotational and action semantics, etc.). A number of programming paradigms can be observed: imperative, declarative, functional, logic, procedural, object-oriented programming. This space is organized around the concepts of program written in a given programming language whose syntax is formally specified in a grammar. What characterizes this space is that the goal is to deal with executable programs.

An important issue in technological spaces is how to handle the interoperability between spaces of different possibilities. A good example of a bridge between two technological spaces is eCommerce, where a constant problem is the difference in XML syntaxes of the exchanged documents; it is typically solved with XSLT transformations. These transformations tend to be complex because several tasks are mixed in XSLT rules: terminology translation, syntactical translation between the XML syntaxes, etc. One proposed solution reduces the complexity by splitting these tasks into different layers [Omelayenko & Fensel, 2001]. The document schemas (e.g., DTDs) are abstracted to a conceptual representation (e.g. RDF Schema). Then the transformation between the two conceptual representations/ontologies is specified. The XSLT transformation can be automatically obtained later. Thus, the translation is specified in a new TS (Ontology engineering). Another example is migration between two programming languages. There are tools which transform Java code directly to C#, but it is possible to create a model from Java code by using UML tool (*reverese engineering*), and then from that model to create C# code (*forward engineering*).

Figure 2.40 shows the MDA technological space which, besides its central MOF modeling space, includes other modeling spaces, e.g., XMI for representation. Transformations to Java or a similar code are, also, models which belong to one or more modeling spaces, and they are included in the MDA technological space. Here exist transformations on the M2 level between certain concepts, while transformations on the M3 level are still subject of [Wimmer & Kramler, 2005] and [Kurtev, 2005].

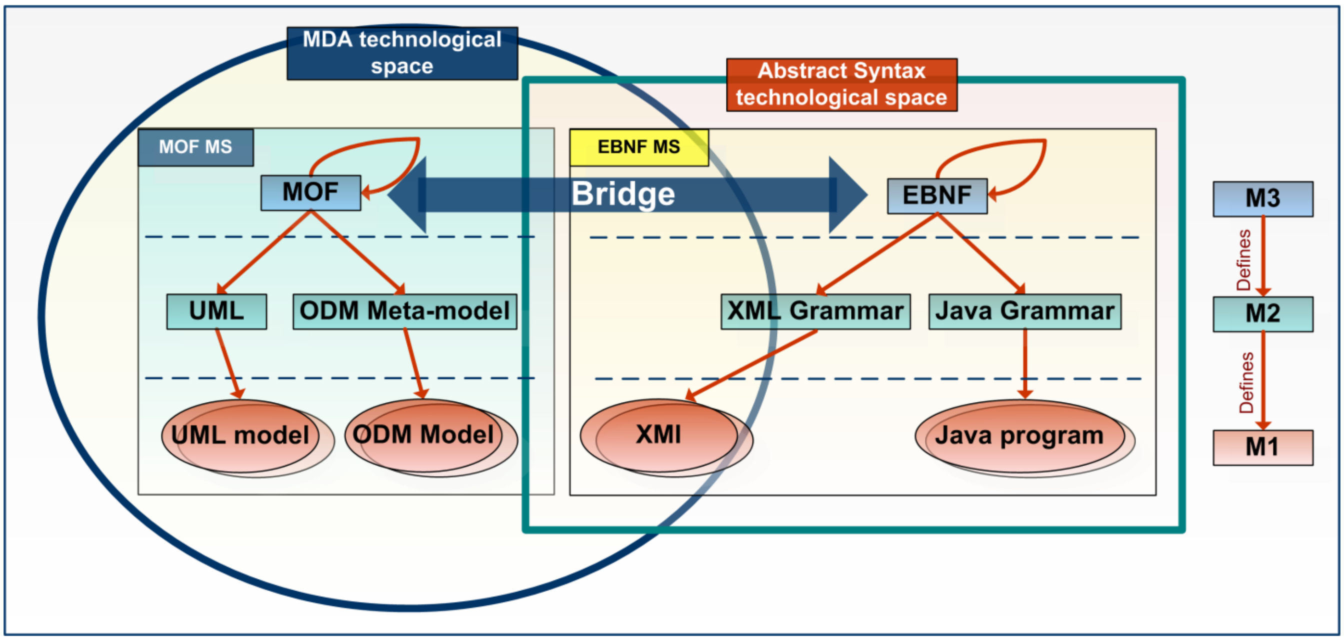


Figure 2.40: MDA TS and Abstract Syntax TS

From this, we can conclude that one technological space include one or more modeling spaces. Every modeling space is part of at least one technological space, while bridge that connects two technological spaces is a medium for their connection.

3. REWERSE I1 RULE MARKUP LANGUAGE AND RULE INTERCHANGE ON THE WEB

It is expected that rule markup languages will be the primary driving force for the widespread use of rules both on the Web and in distributed systems. They allow for deploying, executing, publishing and communicating rules on the Web. They may also play the role of a lingua franca for exchanging rules between different systems and tools. They may be used, for example, to express derivation rules for enriching Web ontologies by adding definitions of derived concepts or for defining data access permissions; to describe and publish the reactive behavior of a system in the form of reaction rules; and to provide a complete XML-based specification of a software agent [Wagner et al., 2006-1]. In a narrow sense, a rule markup language is concrete (XML-based) rule syntax for the Web. In a broader sense, it should have an abstract syntax as a common basis for defining various concrete languages serving different purposes. The main purpose of a rule markup language is to permit reuse, interchange and publication of rules.

3.1. DESIGN OF THE REWERSE I1 RULE MARKUP LANGUAGE (R2ML)

Prof. Dr. Gerd Wagner and Prof. Dr. Adrian Giurca from Institute of Informatics at Brandenburg University of Technology at Cottbus, Germany, have been working on the development of R2ML language prior to the version 0.2. From the version 0.2 to the actual version 0.5 (January 2007), Prof. Dr. Dragan Gašević from Athabasca University in Canada, and the author of this thesis have been involved in the R2ML development, as well.

The chosen approach for rule markup language development is based on Model Driven Architecture [Miller & Mukerji, 2003]. R2ML concrete syntax is defined in the form of an XML Schema. This schema is based on the R2ML MOF-based meta-model. From the perspective of Model Driven Architecture (MDA), rules can be considered at three different abstraction levels (shown in Figure 3.1):

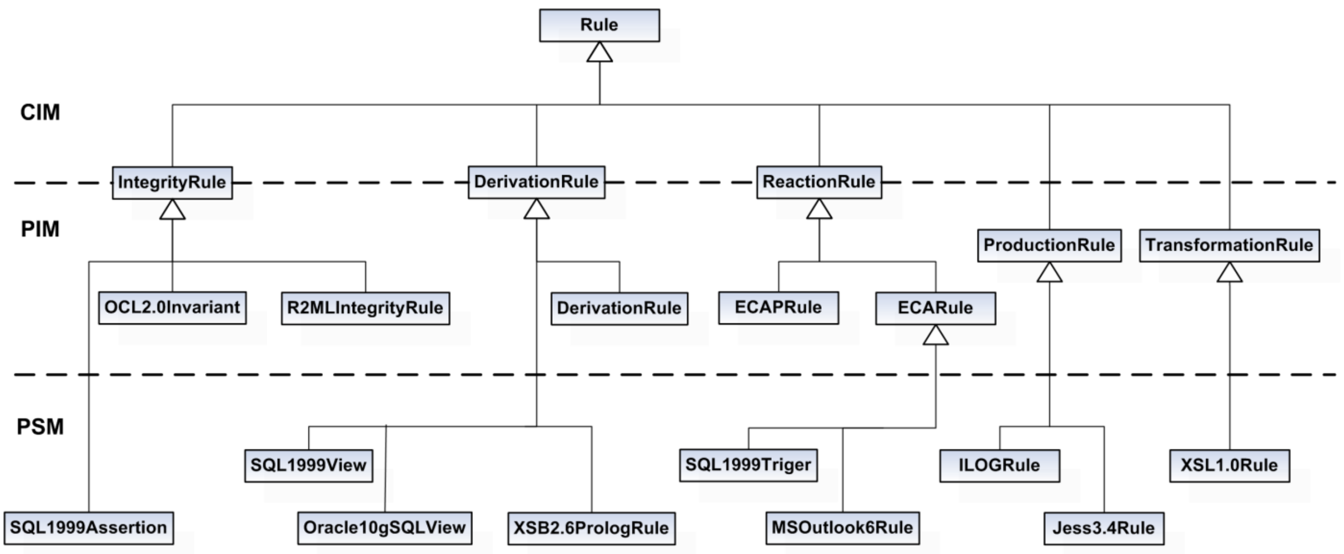


Figure 3.1: The concepts of rules at three different abstraction levels: computation independent (CIM), platform-independent (PIM) and platform-specific (PSM) modeling [Wagner et al., 2006]

At the ('computation-independent') business domain level (CIM), rules are statements that express (certain parts of) a business/domain policy (e.g., defining terms of the domain language) in a declarative manner, typically using a natural language or a visual language. An example of such rule is:

The driver of a rental car must be at least 18 years old. At the platform-independent operational design level (PIM), rules are formal statements, expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software system. Examples of rule languages at this level are SQL:1999 [SQL, 1999] and OCL 2.0 [OCL, 2006]. At the platform-specific implementation level (PSM) rules are statements in a language of a specific execution environment, such as Oracle 10g views, Jess 3.4, XSB 2.6 Prolog or the Microsoft Outlook 6 Rule Wizard.

The R2ML languages try to address all the requirements specified in the Rule Interchange Format (RIF) document [Ginsberg, 2006] in order to provide a general markup language for sharing Web rules. RIF is a W3C initiative that is supposed to define an intermediary language between various rule languages, but is not expected to provide a formally defined semantic foundation for reasoning on the Web, as OWL does for ontologies. The current state of this initiative is that it defines a set of requirements and use cases for sharing rules on the Web. However, there is no official submission to this initiative yet.

3.2. RULES

R2ML can represent different rule constructs, i.e. it captures the most important types of rules. The actual version 0.5 (January 2007) supports the following types of rules: derivation rules, integrity rules (constraints), reaction rules, production rules and transformation rules (in early development).

3.2.1. Integrity rules

Integrity rules in R2ML, also known as (integrity) constraints, consist of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic or OCL (Figure 3.2). R2ML framework supports two kinds of integrity rules: *alethic* and *deontic* integrity rules. An alethic integrity rule can be expressed with a phrase, such as ”*it is necessarily the case that*”, whereas a deontic one can be expressed with phrases, such as ”*it is obligatory that*” or ”*it should be the case that*” [Wagner et al., 2006-1].

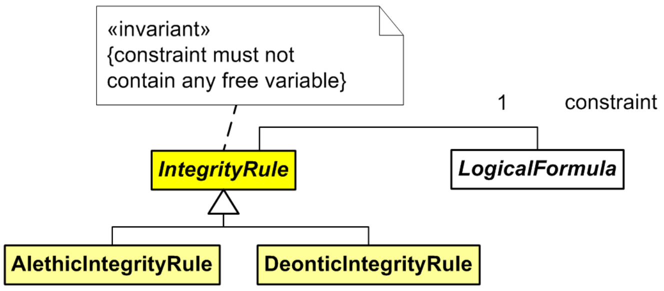


Figure 3.2: Integrity rules in the R2ML meta-model

A constraint assertion is a logical sentence that *must necessarily*, or that *should*, hold in all evolving states and state transition histories of the discrete dynamic system to which it applies. An integrity rule cannot have free variables, i.e. all variables from this formula are quantified. The R2ML XML Schema for integrity rules is shown in Figure 3.3.

```
<!-- LogicalFormula  -->
<xs:complexType name="LogicalFormula" abstract="true"/>

<!-- Abstract Integrity Rule -->
<xs:complexType name="IntegrityRule" abstract="true">
   <xs:complexContent>
      <xs:extension base="r2ml:Rule">
            <xs:sequence>
                  <xs:element ref="r2ml:constraint"/>
            </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<!-- Alethic Integrity Rule -->
<xs:complexType name="AlethicIntegrityRule" abstract="false">
    <xs:complexContent>
         <xs:extension base="r2ml:IntegrityRule"/>
    </xs:complexContent>
</xs:complexType>

<!-- Deontic Integrity Rule -->
<xs:complexType name="DeonticIntegrityRule" abstract="false">
     <xs:complexContent>
          <xs:extension base="r2ml:IntegrityRule"/>
     </xs:complexContent>
</xs:complexType>
```

Figure 3.3: R2ML XML Schema for integrity rules

An example of integrity rule on CIM level is (from the EU-Rent case study[25]): *if rental is not a one way rental then return branch of rental must be the same as pick-up branch of rental*. This rule in R2ML XML concrete syntax is shown in Figure 3.4.

[25] EU-Rent case study at the European Business Rules Conference web site http://www.eurobizrules.org/eurentcs/eurent.htm.

```
<r2ml:AlethicIntegrityRule r2ml:id="IR001">
   <r2ml:constraint>
       <r2ml:UniversallyQuantifiedFormula>
          <r2ml:ObjectVariable r2ml:name="r1" r2ml:classID="Rental"/>
          <r2ml:Implication>
               <r2ml:antecedent>
                    <r2ml:NegationAsFailure>
                         <r2ml:ObjectClassificationAtom r2ml:classID="OneWayRental">
                              <r2ml:ObjectVariable r2ml:name="r1"/>
                         </r2ml:ObjectClassificationAtom>
                    </r2ml:NegationAsFailure>
              </r2ml:antecedent>
              <r2ml:consequent>
                    <r2ml:EqualityAtom>
                         <r2ml:ReferencePropertyFunctionTerm
                          r2ml:referencePropertyID="returnBranch">
                            <r2ml:contextArgument>
                                 <r2ml:ObjectVariable r2ml:name="r1"/>
                            </r2ml:contextArgument>
                         </r2ml:ReferencePropertyFunctionTerm>
                         <r2ml:ReferencePropertyFunctionTerm
                          r2ml:referencePropertyID="pickupBranch">
                            <r2ml:contextArgument>
                                 <r2ml:ObjectVariable r2ml:name="r1"/>
                            </r2ml:contextArgument>
                         </r2ml:ReferencePropertyFunctionTerm>
                   </r2ml:EqualityAtom>
              </r2ml:consequent>
           </r2ml:Implication>
       </r2ml:UniversallyQuantifiedFormula>
   </r2ml:constraint>
</r2ml:AlethicIntegrityRule>
```

Figure 3.4: R2ML XML representation of an integrity rule

3.2.2. Derivation rules

Derivation Rules in R2ML have ”conditions” and ”conclusions” (Figure 3.5). In R2ML language the conditions of a derivation rule are `AndOrNafNegFormula`. Conclusions are restricted to a literal conjunction of atoms (Figure 3.6).

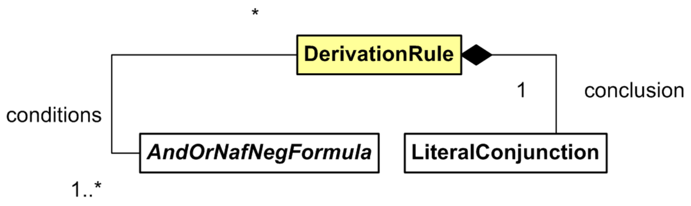


Figure 3.5: Derivation rules in the R2ML meta-model

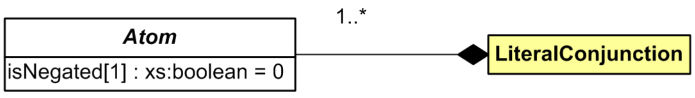


Figure 3.6: LiteralConjuction of R2ML atoms

The R2ML XML Schema for derivation rules is shown in Figure 3.7.

```
<!-- Derivation Rule  -->
<xs:complexType name="DerivationRule" abstract="false">
        <xs:complexContent>
            <xs:extension base="r2ml:Rule">
                 <xs:sequence>
                     <xs:element ref="r2ml:conditions"/>
                     <xs:element ref="r2ml:conclusion"/>
                 </xs:sequence>
            </xs:extension>
        </xs:complexContent>
</xs:complexType>
```

Figure 3.7: R2ML XML Schema for derivation rules

An example of derivation rule on CIM level is: *the discount for a customer buying a product is 7.5 percent if the customer is premium and the product is luxury*. This rule in R2ML XML concrete syntax is shown in Figure 3.8.

```
<r2ml:DerivationRule r2ml:id="DR004">
   <r2ml:conditions>
      <r2ml:ObjectClassificationAtom r2ml:classID="PremiumCustomer">
         <r2ml:ObjectVariable r2ml:name="customer" r2ml:classID="Customer"/>
      </r2ml:ObjectClassificationAtom>
      <r2ml:ObjectClassificationAtom r2ml:classID="LuxuryProduct">
         <r2ml:ObjectVariable r2ml:name="product" r2ml:classID="Product"/>
      </r2ml:ObjectClassificationAtom>
      <r2ml:AssociationAtom r2ml:associationPredicateID="buy">
         <r2ml:objectArguments>
            <r2ml:ObjectVariable r2ml:name="customer"/>
               <r2ml:ObjectVariable r2ml:name="product"/>
         </r2ml:objectArguments>
      </r2ml:AssociationAtom>
   </r2ml:conditions>
   <r2ml:conclusion>
      <r2ml:AttributionAtom r2ml:attributeID="discount">
         <r2ml:subject>
            <r2ml:ObjectVariable r2ml:name="customer"/>
         </r2ml:subject>
         <r2ml:value>
            <r2ml:TypedLiteral r2ml:datatype="xs:decimal"
              r2ml:lexicalValue="7.5"/>
         </r2ml:value>
      </r2ml:AttributionAtom>
   </r2ml:conclusion>
</r2ml:DerivationRule>
```

Figure 3.8: R2ML XML representation of a derivation rule

3.2.3. Production rules

Production rules in R2ML have ”conditions” and ”post-conditions” (Figure 3.9). The conditions and post-conditions of a production rule are `AndOrNafNegFormula`. A production rule may execute an `Action`, as shown in Figure 3.31. While OCL can be used in a platform-independent production rule language to specify conditions on an object-oriented system state, the UML Action Semantics can be used to specify produced actions.

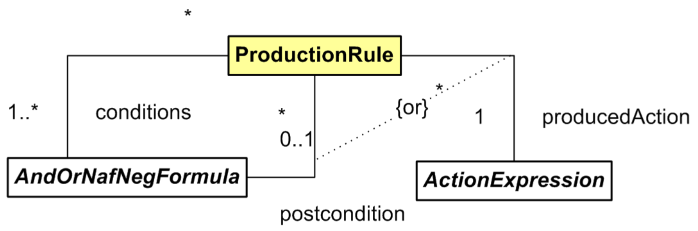


Figure 3.9: Production rules in the R2ML meta-model

The R2ML XML Schema for production rules is shown in Figure 3.10.

```
<!-- Production Rule  -->
<xs:complexType name="ProductionRule" abstract="false">
     <xs:complexContent>
          <xs:extension base="r2ml:Rule">
               <xs:sequence>
                    <xs:element ref="r2ml:conditions"/>
                    <xs:element ref="r2ml:producedAction"/>
                    <xs:element ref="r2ml:postcondition"
                                minOccurs="0"/>
               </xs:sequence>
          </xs:extension>
      </xs:complexContent>
</xs:complexType>
```

Figure 3.10: R2ML XML Schema for production rules

An example of a production rule on the CIM level is: *if the order value is greater than 1000 and the customer type is not gold then give a 10% discount*. This rule in R2ML XML concrete syntax is shown in Figure 3.11.

```
<r2ml:ProductionRule r2ml:id="PR001" >
     <r2ml:conditions>
           <r2ml:qf.Conjunction>
              <r2ml:DataPredicateAtom r2ml:dataPredicateID="swrlb:greaterThan">
                <r2ml:dataArguments>
                     <r2ml:AttributeFunctionTerm r2ml:attributeID="orderValue">
                        <r2ml:contextArgument>
                              <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
                        </r2ml:contextArgument>
                     </r2ml:AttributeFunctionTerm>
                     <r2ml:TypedLiteral r2ml:lexicalValue="1000" r2ml:type="xs:positiveInteger"/>
                </r2ml:dataArguments>
              </r2ml:DataPredicateAtom>
              <r2ml:DataPredicateAtom r2ml:dataPredicateID="swrlb:equal" r2ml:isNegated="true">
                 <r2ml:dataArguments>
                      <r2ml:AttributeFunctionTerm r2ml:attributeID="customerRating">
                        <r2ml:contextArgument>
                              <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
                        </r2ml:contextArgument>
                      </r2ml:AttributeFunctionTerm>
                      <r2ml:TypedLiteral r2ml:lexicalValue="gold" r2ml:type="xs:string"/>
                 </r2ml:dataArguments>
              </r2ml:DataPredicateAtom>
           </r2ml:qf.Conjunction>
     </r2ml:conditions>
     <r2ml:producedAction>
            <r2ml:AssignActionExpression r2ml:propertyID="srv:discount">
              <r2ml:contextArgument>
                  <r2ml:ObjectVariable r2ml:name="order" r2ml:classID="srv:Order"/>
              </r2ml:contextArgument>
              <r2ml:TypedLiteral r2ml:lexicalValue="10" r2ml:type="xs:positiveInteger"/>
            </r2ml:AssignActionExpression>
     </r2ml:producedAction>
</r2ml:ProductionRule>
```

Figure 3.11: R2ML XML representation of a production rule

3.2.4. Reaction rules

Reaction rules consist of a mandatory triggering event expression, an optional condition, and a produced action or a post-condition (or both), which are roles of type `EventExpression`, `AndOrNafNegFormula`, `ActionExpression`, and `AndOrNafNegFormula`, respectively, as shown in Figure 3.12. While the condition of a reaction rule is, exactly like the condition of a derivation rule, a quantifier-free formula, the post-condition is restricted to a conjunction of possibly negated atoms. There are two types of reaction rules: those that do not have a post-condition, which are the well-known *Event-Condition-Action (ECA)* rules, and those that do have a postcondition, which we call *ECAP rules*.

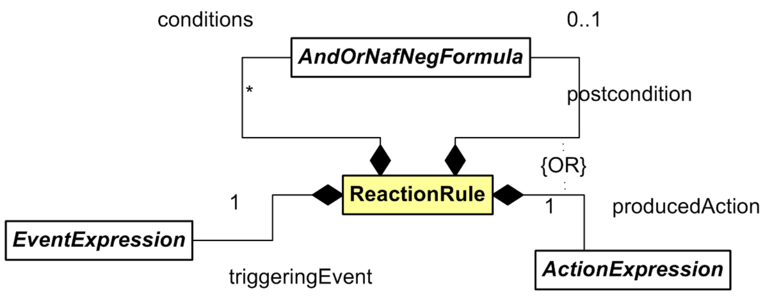


Figure 3.12: Reaction rules in the R2ML meta-model

A reaction rule consists of the following components:

- `triggeringEvent` is an R2ML `EventExpression`, which is either atomic or composite.

- `conditions` are represented as a collection of `AndOrNafNegFormula`, and as `postcondition`.
- `producedAction` is an R2ML action, that represents a condition change in a system. The actual version of R2ML (0.5) defines composite actions, like sequential or parallel actions.

The R2ML XML Schema for reaction rules is shown in Figure 3.13.

```
<!-- Reaction Rule  -->
<xs:complexType name="ReactionRule" abstract="false">
     <xs:complexContent>
           <xs:extension base="r2ml:Rule">
                 <xs:sequence>
                        <xs:element ref="r2ml:triggeringEvent"/>
                        <xs:element ref="r2ml:conditions" minOccurs="0"/>
                        <xs:element ref="r2ml:producedAction"/>
                        <xs:element ref="r2ml:postcondition" minOccurs="0"/>
                 </xs:sequence>
           </xs:extension>
     </xs:complexContent>
</xs:complexType>
```

Figure 3.13: R2ML XML Schema for reaction rules

An example of reaction rule on CIM level is: *if customer returns a car and the car has more than 5000km from the last service then send the car to the service*. This rule in R2ML XML concrete syntax is shown in Figure 3.14.

```
<r2ml:ReactionRule r2ml:id="ECA001">
      <r2ml:triggeringEvent>
            <r2ml:MessageEventExpression r2ml:eventType="alert"
                                         ml:startTime="2006-03-21T09:00:00"
                                         ml:duration="P0Y0M0DT0H0M0S"
                                         ml:sender="http://www.mywebsite.org">
                  <r2ml:arguments>
                        <r2ml:ObjectVariable r2ml:name="car" r2ml:class="RentalCar"/>
                        <r2ml:ObjectVariable r2ml:name="customer" r2ml:class="Customer"/>
                  </r2ml:arguments>
            </r2ml:MessageEventExpression>
      </r2ml:triggeringEvent>
      <r2ml:conditions>
        <r2ml:DatatypePredicateAtom r2ml:datatypePredicate="ge">
              <r2ml:dataArguments>
                  <r2ml:AttributeFunctionTerm r2ml:attribute="srv:lastservice">
                    <r2ml:contextArgument>
                        <r2ml:ObjectVariable r2ml:name="rentalCar" r2ml:class="srv:RentalCar"/>
                    </r2ml:contextArgument>
                  </r2ml:AttributeFunctionTerm>
                  <r2ml:AttributeFunctionTerm r2ml:attribute="odometer_reading">
                    <r2ml:contextArgument>
                        <r2ml:ObjectVariable r2ml:name="rentalCar" r2ml:class="srv:RentalCar"/>
                    </r2ml:contextArgument>
                  </r2ml:AttributeFunctionTerm>
                  <r2ml:TypedLiteral r2ml:datatype="xs:positiveInteger" r2ml:lexicalValue="5000"/>
              </r2ml:dataArguments>
         </r2ml:DatatypePredicateAtom>
       </r2ml:conditions>
       <r2ml:producedAction>
            <r2ml:InvokeActionExpression r2ml:operation="service">
                  <r2ml:contextArgument>
                         <r2ml:ObjectVariable r2ml:name="rentalCar" r2ml:class="srv:RentalCar"/>
                  </r2ml:contextArgument>
            </r2ml:InvokeActionExpression>
       </r2ml:producedAction>
</r2ml:ReactionRule>
```

Figure 3.14: R2ML XML representation of a reaction rule

3.3 RULE LANGUAGE META-MODELS

In this section, we describe abstract syntaxes (i.e., meta-models) of rule languages. R2ML, RuleML, SWRL and OCL MOF-based meta-models are shown.

3.3.1. R2ML meta-model

The R2ML meta-model is defined by using MOF modeling language [Wagner et al., 2006]. In the previous section, we showed R2ML class diagrams for different types of rules that R2ML support.

3.3.1.1. Basic constructs of R2ML vocabulary

R2ML language has a basic vocabulary that is defined to support basic rule constructs:

- Vocabulary for classification (basic): `Vocabulary`, `VocabularyEntry`, `Predicate`, `Property`, `Type`, `DatatypePredicate`, `Attribute`, `Class`, `Datatype` and `ObjectName`.
- Vocabulary for functional constructs[26]: `EnumerationDatatype`, `GenericFunction`, `DatatypeFunction`, `Operation`, `DataOperation` and `ObjectOperation`.

[26] Functional constructs describe functional characteristics in a vocabulary; they represent operations that can be executed by some entity or can be used to translate some vocabulary elements (set) into other elements (set).

- Vocabulary for relational constructs[27]: `GenericPredicate` and `AssocationPredicate`.

Figure 3.15 shows vocabulary in the R2ML meta-model. All previously described constructs are shown in this figure (blue classes are of the R2ML vocabulary, strong yellow classes are abstract classes and light yellow are concrete classes).

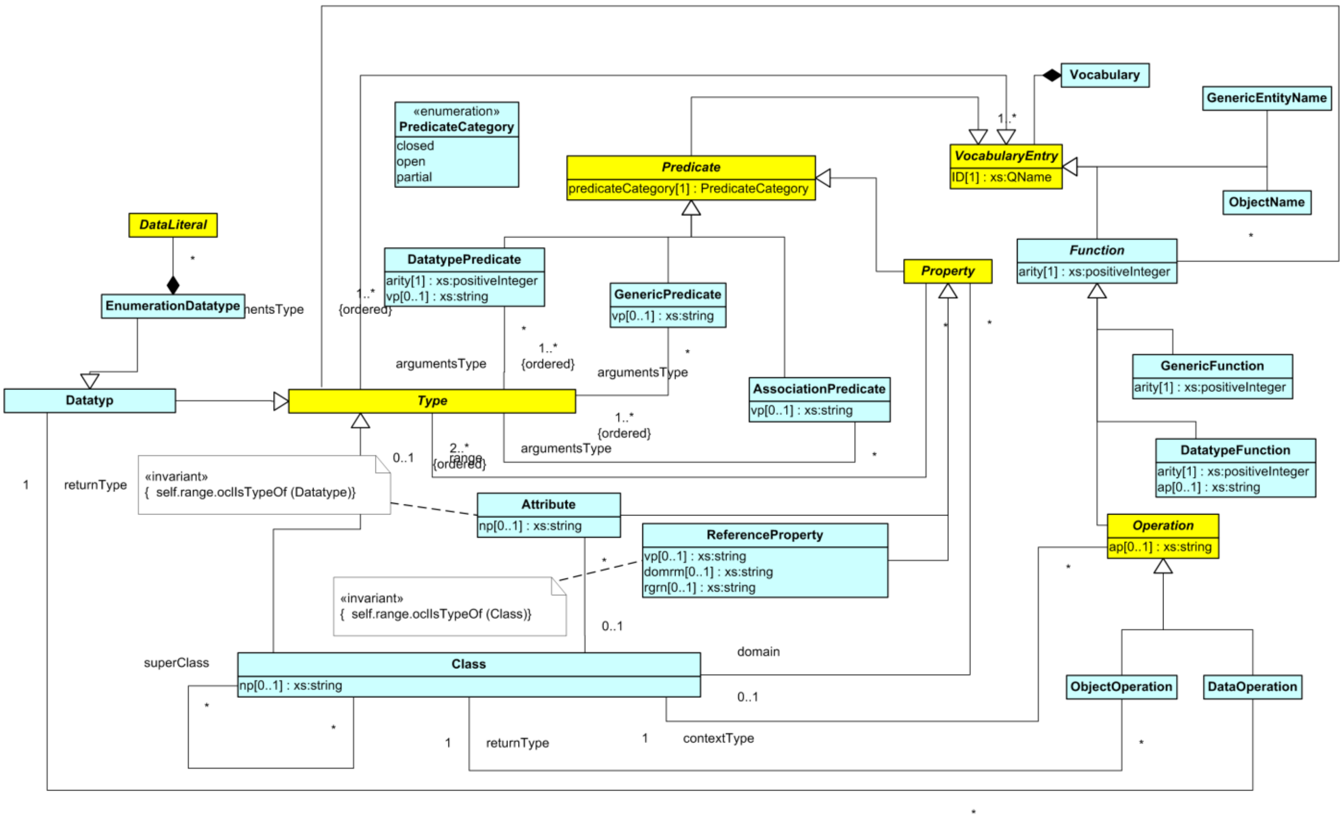


Figure 3.15: Vocabulary in the R2ML meta-model

3.3.1.2. Objects, Data, Variables

R2ML have three types of terms: `GenericTerm`, `ObjectTerm` and `DataTerm`. The concept of `ObjectTerm`, shown in Figure 3.16 is used for modeling variables that can be instantiated by object values and object constants. `ReferencePropertyFunctionTerm` is an object term that is used to model an association end. `ObjectOperationTerm` is an object term that is used to model an operation on contextual argument.

[27] Relational constructs defines relations between one or more vocabulary elements - if exists some relation between two vocabulary elements, then certain predicate will have *true* value. In opposite it will be *false*.

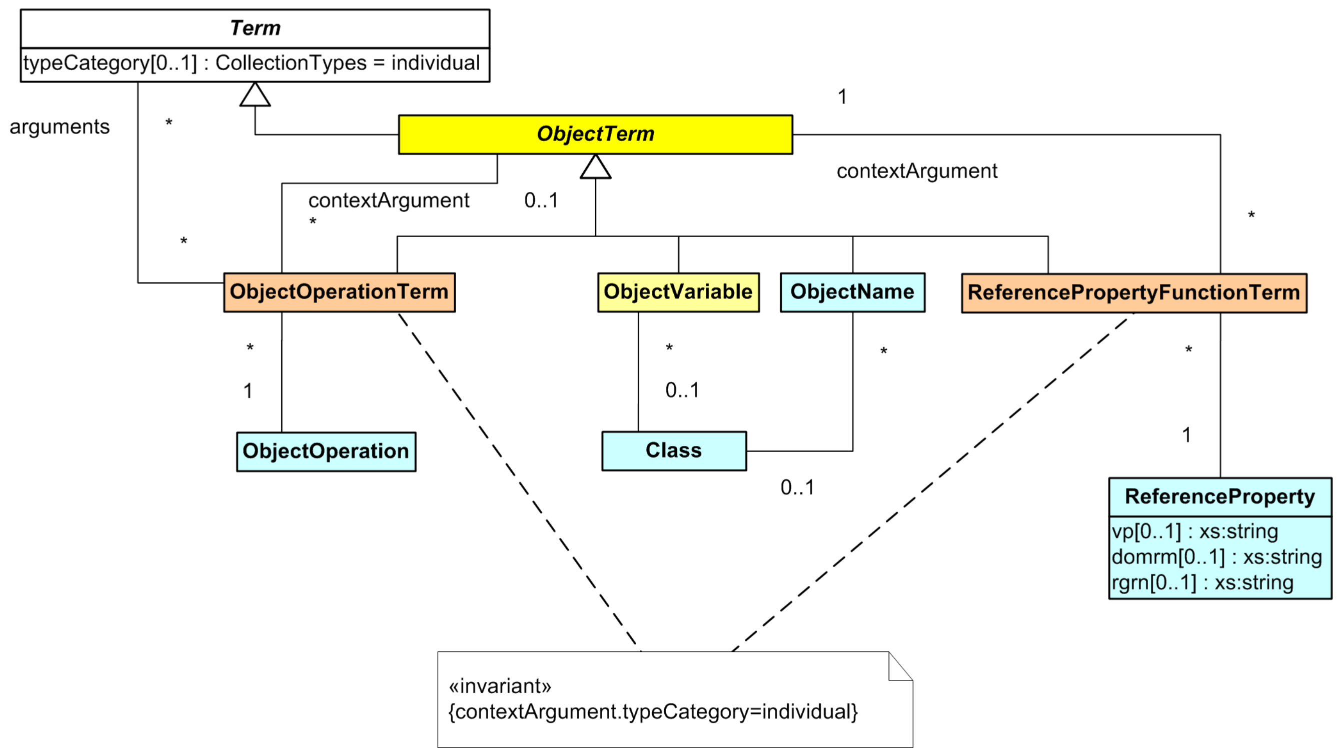


Figure 3.16: `ObjectTerm` in the R2ML meta-model

`DataTerm` is used to represent primitive data types and data values (Figure 3.17). There are three types of data terms: `DataVariable`, which represents a variable, `DataLiteral`, which represents a value and `DataFunctionTerm`. `DataFunctionTerm` can be of three different types:

- `DatatypeFunctionTerm` represents arithmetic built-ins;
- `AttributeFunctionTerm` represents an attribute function (a function, which returns attribute value for an object);
- `DataOperationTerm` represents a user-defined function (method, for instance) and takes `DataTerm` or `ObjectTerm` as parameters.

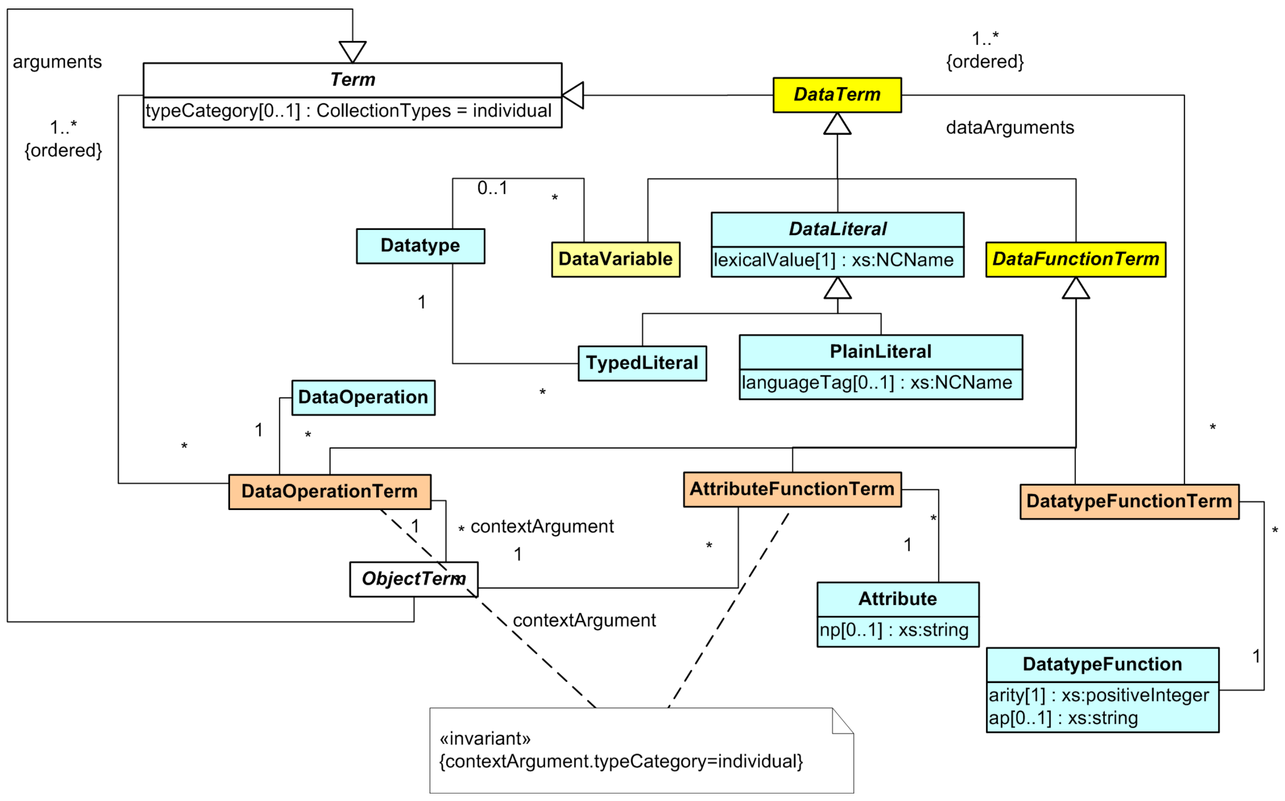


Figure 3.17: DataTerm in the R2ML meta-model

GenericTerm (Figure 3.18) is used for modeling variables that may, or may not, have type (GenericVariable), constants (GenericEntityName), as well as generic functions (GenericFunctionTerm).

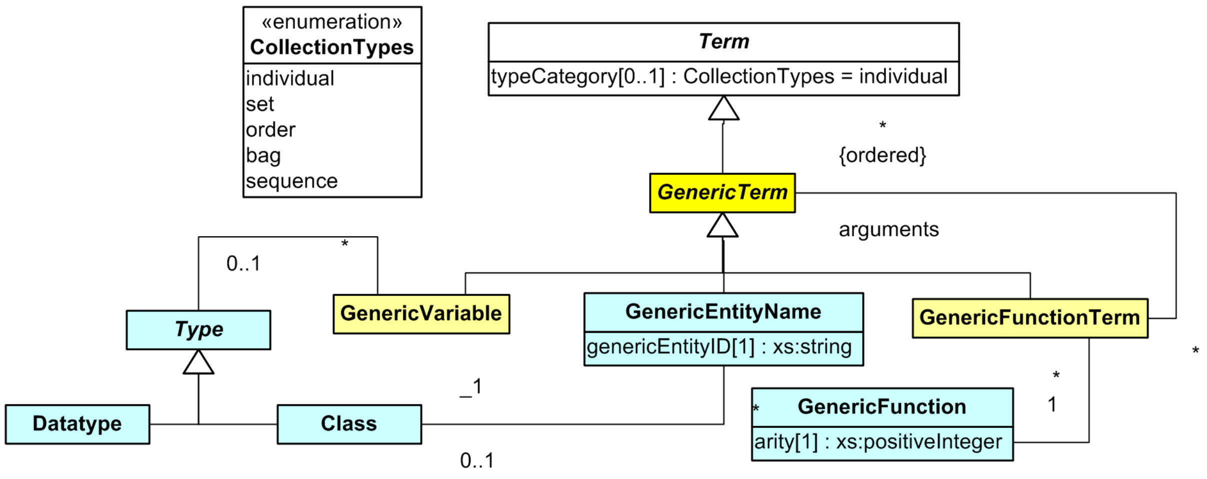


Figure 3.18: GenericTerm in the R2ML meta-model

R2ML supports three types of variables: GenericVariable, ObjectVariable and DataVariable (Figure 3.19).

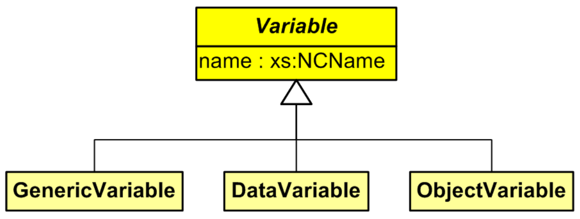


Figure 3.19: Variables in the R2ML meta-model

3.3.1.3. Atoms

The basic constituent of a rule is an atom. R2ML defines a meta-model for atoms, which is compatible with all important concepts of OWL, SWRL and RuleML. All atoms from the R2ML meta-model are presented on Figure 3.20.

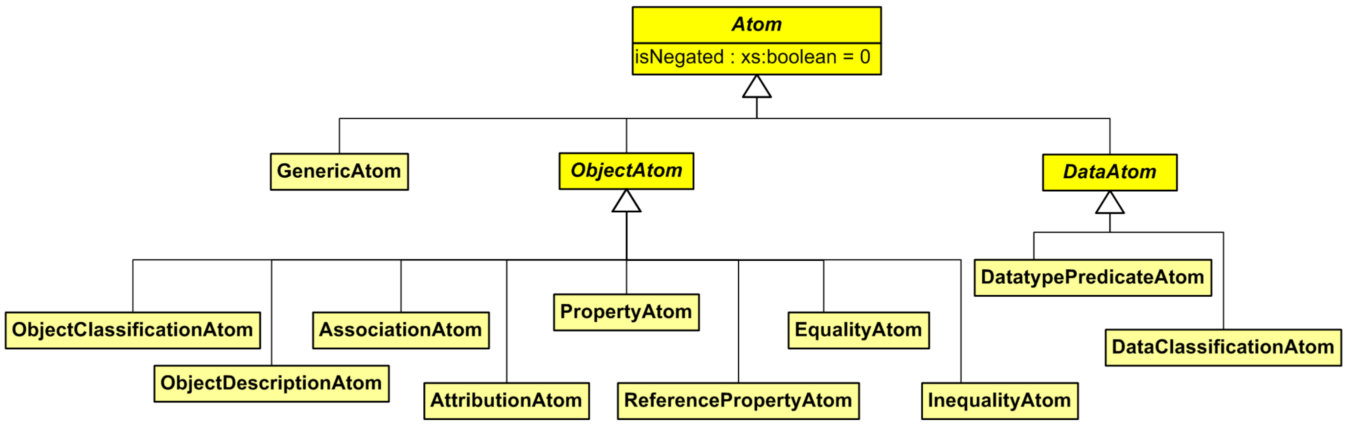


Figure 3.20: Atoms in the R2ML meta-model

An object classification atom (Figure 3.21) refers to a class and consists of an object term.

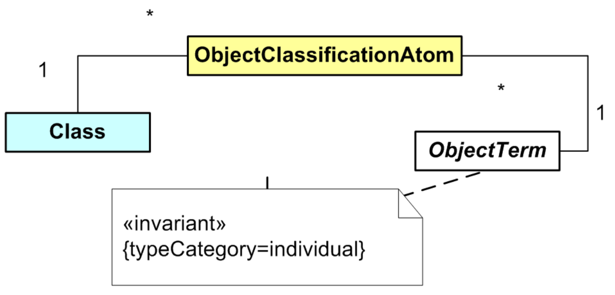


Figure 3.21: `ObjectClassificationAtom` in the R2ML meta-model

An object description atom (Figure 3.22) refers to a class as a base type, and to zero or more classes as categories. It consists of a number of slots (attribute data slot and reference property object slot). An instance of such atom refers to one particular object.

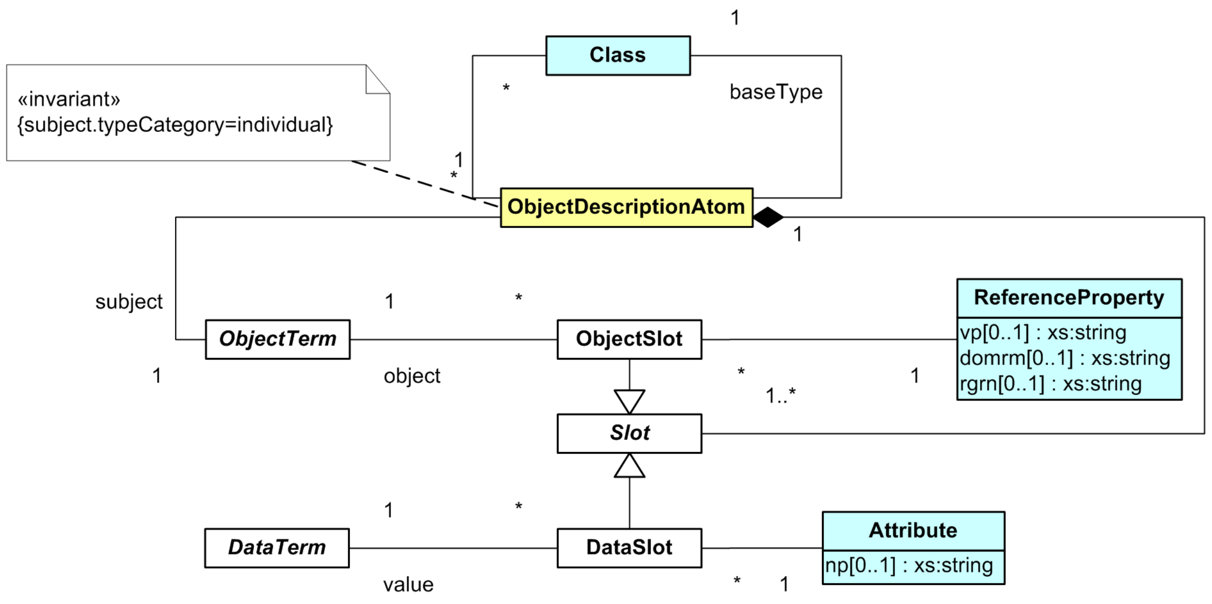


Figure 3.22: `ObjectDescriptionAtom` in the R2ML meta-model

An attribution atom (Figure 3.23) consists of an object term as ”subject”, and a data term as ”value”.

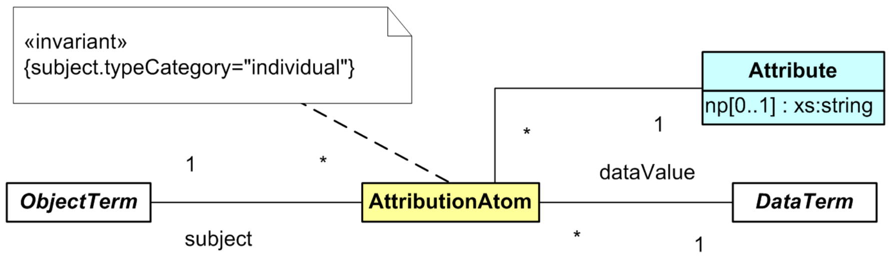


Figure 3.23: `AttributionAtom` in the R2ML meta-model

A reference property atom (Figure 3.24) associates object terms as ”subjects” with other object terms as ”objects”.

In order to directly support common fact types of natural language, it is important to have n-ary predicates (for n > 2).

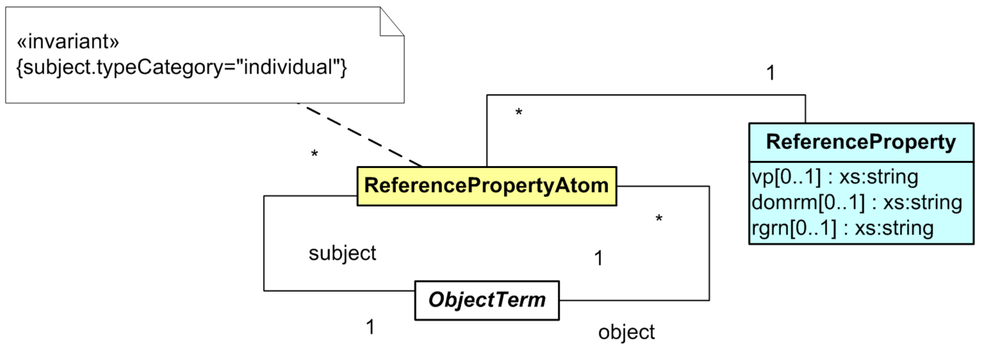


Figure 3.24: `ReferencePropertyAtom` in the R2ML meta-model

An association atom (Figure 3.25) is constructed using an n-ary predicate as association predicate, a collection of data terms as ”data arguments”, and a collection of object terms as ”object arguments”.

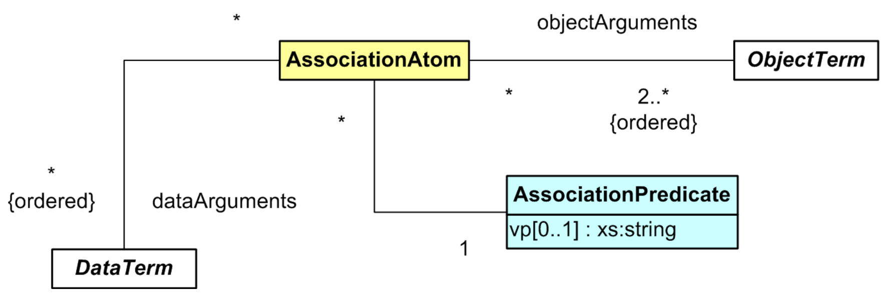


Figure 3.25: `AssociationAtom` in the R2ML meta-model

Both equality atom and inequality atom (Figure 3.26) are composed of two or more object terms.

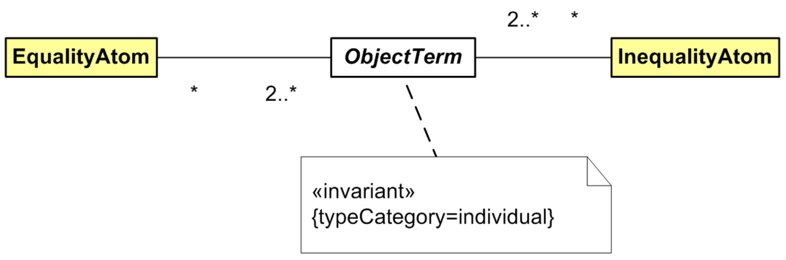


Figure 3.26: `EqualityAtom` and `InequalityAtom` in the R2ML meta-model

A data classification atom (Figure 3.27) consists of a data term and refers to a data type.

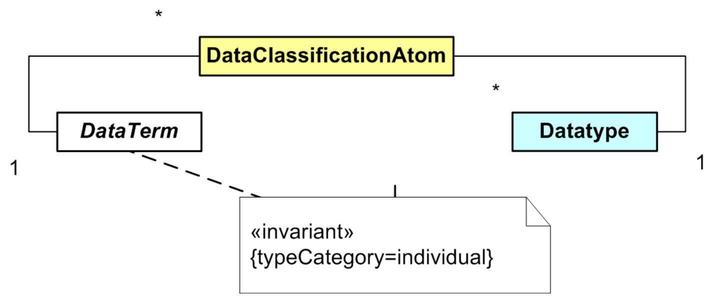


Figure 3.27: `DataClassificationAtom` in the R2ML meta-model

`GenericAtom` consists of a predicate (which can be of type: `ObjectClassificationPredicate`, `AttributionPredicate`, `AssociationPredicate`, `ReferencePropertyPredicate`, `EqualityPredicate`, `InequalityPredicate`, `DatatypePredicate` and `DataClassificationPredicate`) and arguments, as shown in Figure 3.28.

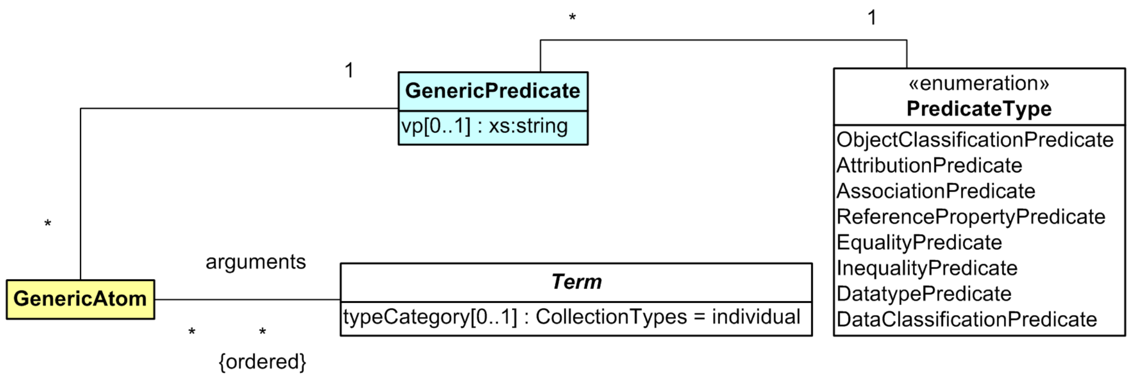


Figure 3.28: `GenericAtom` in the R2ML meta-model

3.3.1.4. Formulas

R2ML provides two abstract concepts for formulas: the concept of `AndOrNafNegFormula` (Figure 3.29), which represents the most general quantifier-free logical formula with weak and strong negations, and the concept of `LogicalFormula` (Figure 3.30), which corresponds to a general first order formula.

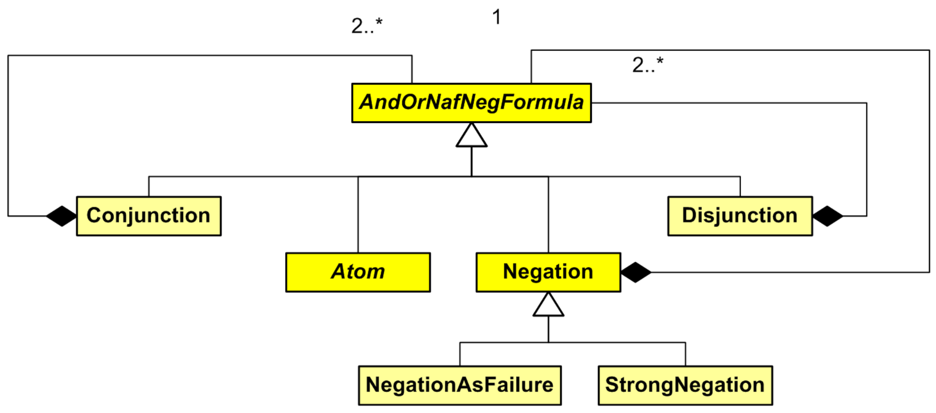


Figure 3.29: `AndOrNafNegFormula` in the R2ML meta-model

R2ML supports two kinds of negation (as shown in Figure 3.30). The distinction between weak and strong negation is used in several computational languages (like SQL [SQL, 1999] and OCL [OCL, 2006]), and it is presented in [Wagner, 2003]. Weak negation captures the absence of positive information, while strong negation captures the presence of explicit negative information. Weak negation captures the computational concept of negation-as-failure (or closed-world negation).

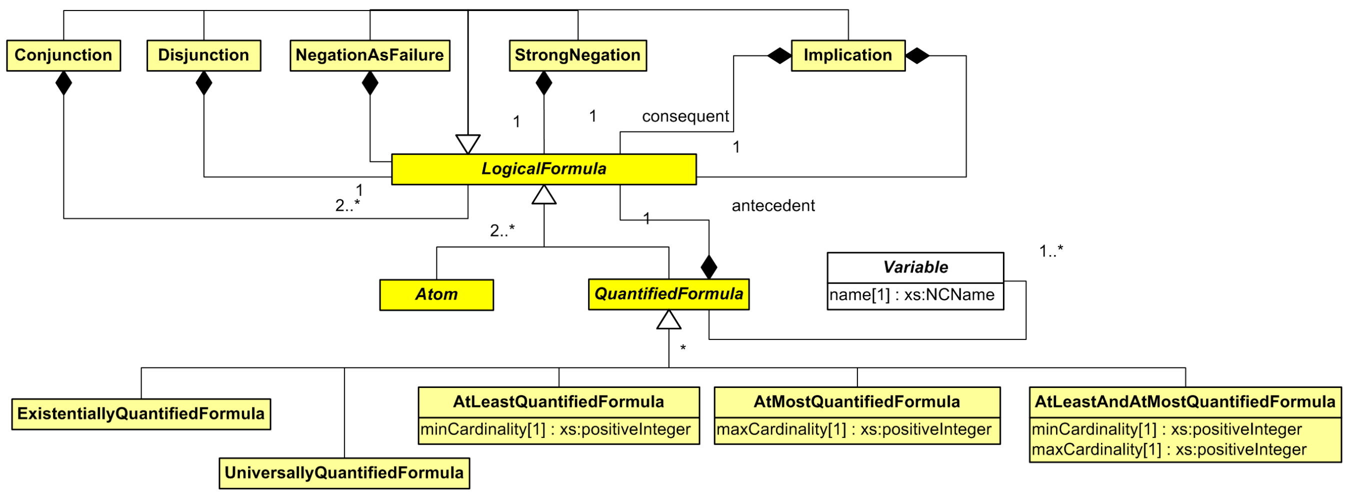

Conjunction
Disjunction
NegationAsFailure
StrongNegation
Implication
consequent
antecedent
LogicalFormula
Atom
QuantifiedFormula
Variable
name[1] : xs:NCName
ExistentiallyQuantifiedFormula
UniversallyQuantifiedFormula
AtLeastQuantifiedFormula
minCardinality[1] : xs:positiveInteger
AtMostQuantifiedFormula
maxCardinality[1] : xs:positiveInteger
AtLeastAndAtMostQuantifiedFormula
minCardinality[1] : xs:positiveInteger
maxCardinality[1] : xs:positiveInteger
2..*
1..*

Figure 3.30: `LogicalFormula` in the R2ML meta-model

3.3.1.5. Actions

R2ML supports five types of actions: `InvokeAction`, `AssignAction`, `CreateAction`, `DeleteAction` and `SOAPAction` (Figure 3.31).

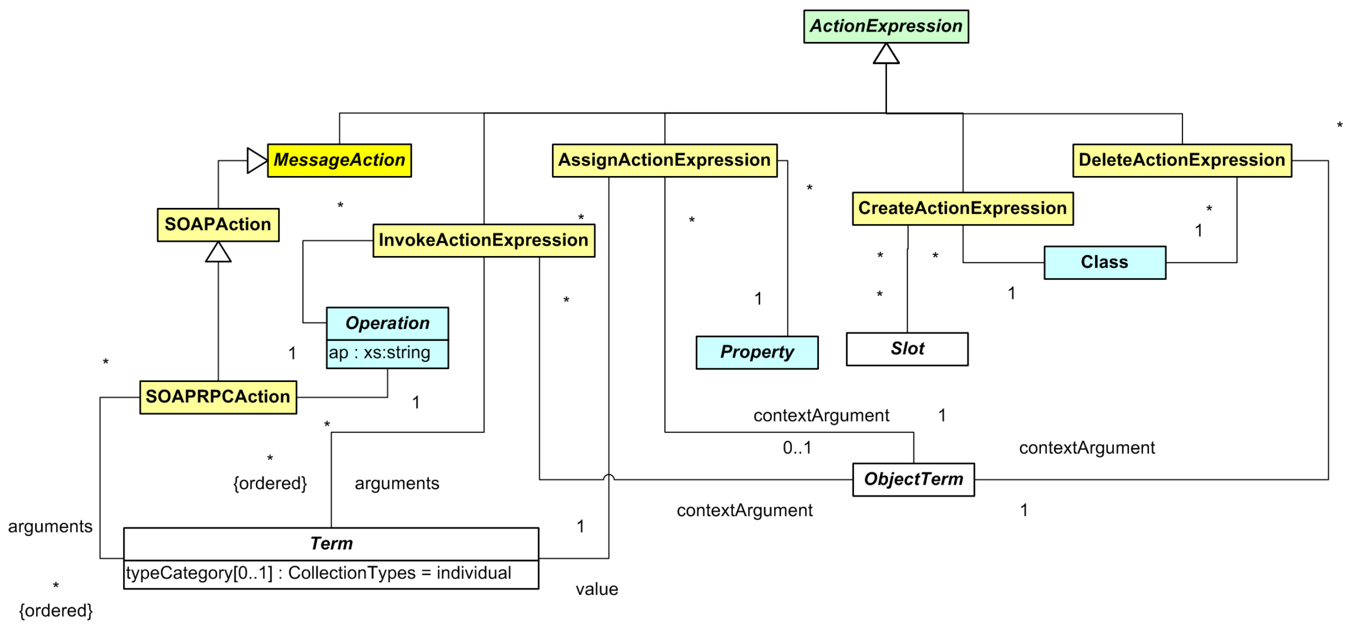


Figure 3.31: Actions in the R2ML meta-model

- `InvokeAction` refers to an UML `Operation` and contains a list of arguments. This action invokes an operation with a list of arguments.
- `AssignAction` refers to an UML `Property` and contains a `DataTerm`. This action assigns a value to a property.
- `CreateAction` refers to an UML `Class` and to an UML `InputPin` as a parameter.
- `DeleteAction` refers to an UML `Class` and contains an `ObjectVariable`. This action removes an instance of the `Class`.
- `SOAPAction` refers to the way that certain Web service is called with SOAP message.

3.3.2. RuleML meta-model

The RuleML abstract syntax (i.e., meta-model) is defined by using the MOF modeling language, which means that it has the same foundation as R2ML. A proposal for RuleML meta-model (for version 0.8) is given in [Wagner et al., 2003]. The actual version of RuleML is 0.91[28]. RuleML supports the same rule types as R2ML, i.e., integrity, derivation, reaction, production and transformation rules.

3.3.2.1. Integrity rules

Like R2ML, RuleML supports integrity rules (also known as integrity constraints). A rule of this type consists of a single logical sentence. It expresses an assertion that must hold in all evolving states and state transitions histories of the discrete dynamic system for which it is defined. An example of this type of rule on CIM level is: "*The confirmation of a rental reservation must lead to an allocation of a car of the requested car group, for the requested date prior to the given date*".

Well-known languages for expressing constraint rules are SQL and OCL. In logic programming, rules with empty heads are sometimes used as constraint rules.

3.3.2.2. Derivation rules

Same as in R2ML, derivation rules in RuleML consist of one or more conditions and a conclusion, both of the `LogicalFormula` type. For specific types of derivation rules, such as definite Horn clauses, the types of condition and conclusion are specifically restricted.

In RuleML meta-model, a derivation rule has two roles, condition (`_Condition`) and conclusion (`_Conclusion`). The latter is an automatic predicate logic formula (of type `AtomicPLFormula`), whereas the former is a, conjunctively interpreted, set of such formulas, as shown in Figure 3.32.

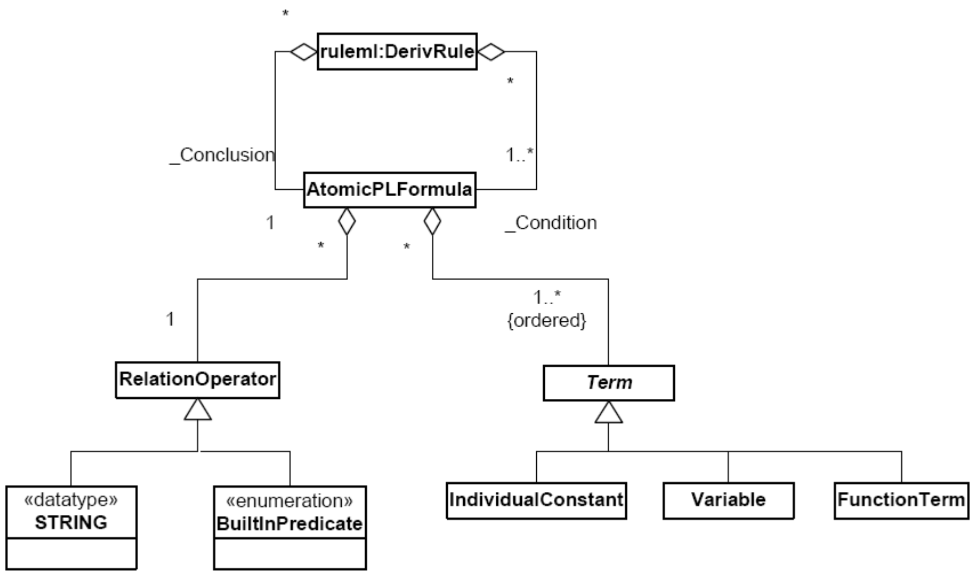


Figure 3.32: Derivation rules in the RuleML meta-model [Wagner et al., 2003]

Concrete syntax of this type of rule conforms to the actual version of RuleML (0.91). In RuleML XML-based concrete syntax, head stands for the conclusion, and body for the condition part. One kind of negation in RuleML is expressed by the negation-as-failure tag (`<Naf>`).

[28] The Rule Markup Initiative: http://www.ruleml.org/.

An example of derivation rule on CIM level is: "*A car is available for rental if it is not assigned to any rental contract and does not require service*". This rule in RuleML XML-based concrete syntax is shown in Figure 3.33.

```
<Implies>
     <head>
           <Atom>
             <op>
                 <Rel>isAvailable</Rel>
             </op>
             <Var>Automobile</Var>
           </Atom>
      </head>
      <body>
            <Atom>
              <op>
                 <Rel>isForRent</Rel>
              </op>
              <Var>Automobile</Var>
            </Atom>
            <Naf>
               <Atom>
                   <op>
                      <Rel>requiresService</Rel>
                   </op>
                   <Var>Automobile</Var>
               </Atom>
            </Naf>
            <Naf>
               <Atom>
                    <op>
                       <Rel>isAssignedToContract</Rel>
                    </op>
                    <Var>Automobile</Var>
               </Atom>
            </Naf>
     </body>
</Implies>
```

Figure 3.33: RuleML XML representation of a derivation rule

3.3.2.3. Reaction rules

Reaction rules in the RuleML meta-model consist of a triggering event, a condition, a triggered action, and a possible post-condition, which are roles of types `EventTerm`, `LogicalFormula`, `ActionTerm`, and `LogicalFormula`, respectively, as shown in Figure 3.34. This type of rules is considered to be the most important type of business rules [Taveter & Wagner, 2001].

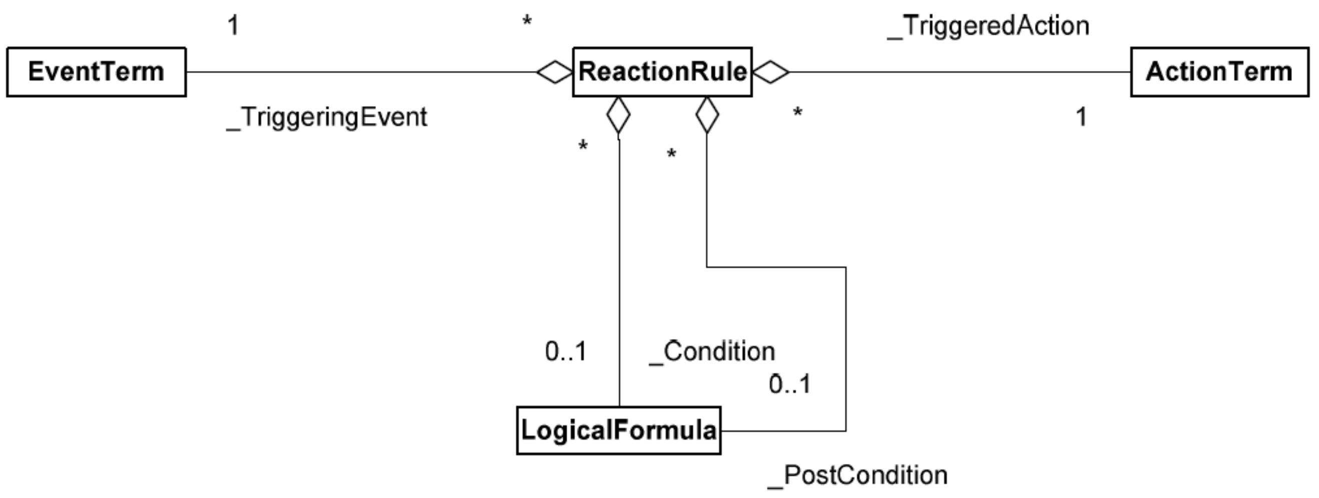


Figure 3.34: Reaction rules in the RuleML meta-model

Reaction rules are basic rules in the RuleML family of languages. They can only be applied in the forward direction in natural fashion, observing/checking events/conditions and performing an action if and when all events/conditons have been perceived or fulfilled.

The UML Action Semantics can be used to specify triggered actions in a platform-independent manner.

There are two types of reaction rules in RuleML: those that do not have a postcondition, which are known as *Event-Condition-Action* (ECA) rules, and those that do have a postconditon, which are called *Event-Condition-Action-Postcondition* (ECAP) rules. ECA rules are now called *database triggers* in SQL. They refer exclusively to database state change events, and do not allow referring to general events. An application-specific ECA rule language may be used in software applications for handling application events in an automated fashion. A well-known example of this is Microsoft Outlook rule wizard (filter), which allows one to specify email handling rules referring to incoming or outgoing message events. ECAP rules extend ECA rules by adding a postcondition that accompanies the triggered action. ECAP rules allow for specifying the effect of a triggered action on the system state in a declarative manner, instead of specifying this state change procedurally by means of corresponding (SQL) write operations. An example of an ECAP rule on CIM level is: "*Upon receiving an invoice requesting a payment of x USD, if the invoice is correct and the balance on the account is y and y is greater than x, then make a payment of x USD resulting in a balance of y - x*".

3.3.2.4. Production rules

Production rules in RuleML consist of a condition and a produced action, which are roles of the type `LogicalFormula` and `ActionTerm`, respectively, as shown in Figure 3.35. Production rules do not explicitly refer to events, but events can be simulated in a production rule system by externally asserting corresponding facts into the working memory. In this way, production rules can implement reaction rules.

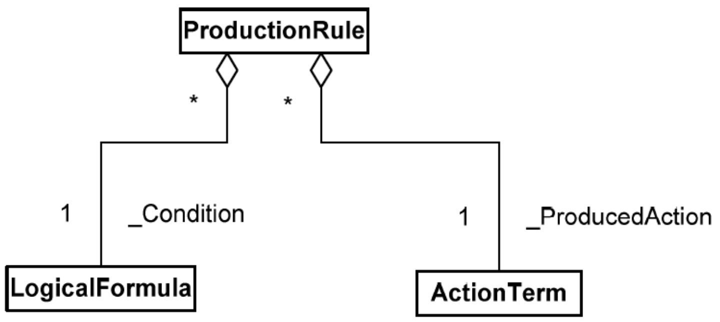


Figure 3.35: Production rules in the RuleML meta-model

Production rules can also implement derivation rules. A derivation rule can be simulated by a production rule of the form if-*Condition*-then-assert-*Conclusion* using the special action *assert* that changes the state of a production rule system by adding a new fact to the set of available facts. These rules are most widely used in business rules industry. Well-known examples of production rules systems are: JESS, iLOG Rules/JRules, Fair Isaas/Blaze Advisor, ART*Enterprise, CA Aion, Haley and ESI Logist.

3.3.2.5. Transformation rules

Transformation rules in RuleML consist of a transformation invoker, a condition, and a transformation return, which are roles of the type `FunctionalFormula`, `LogicalFormula` and `FunctionalFormula`, respectively, as shown in Figure 3.36.

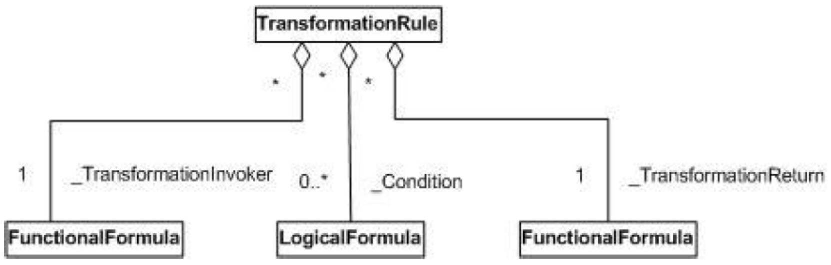


Figure 3.36: Transformation rules in the RuleML meta-model

`LogicalFormula` element is the same as in derivation rules. `FunctionalFormula` applies a `FunctionOperator` to ordered arguments much like an `AtomicPLFormula` does for a `RelationalOperator`.

These types of rules were introduced in RuleML 0.81. While transformation rules generally have the same concrete syntax as other rule types (specifically for the condition part), they also introduce new constructs. The `<trans>` element is the top-level element denoting a transformation rule. It uses the transformation invoker role `_headf` ("head of functions"), optionally followed by a condition role `_body`, which is in turn followed by the transformation role `_foot`.

An example of RuleML transformation rule on CIM level is: "*Given a full description of a book, transform it into a short description containing only the author name and book title*". An example of this rule in RuleML XML-based concrete syntax is shown in Figure 3.37.

```
<rulebase direction="bidirectional">
  <trans>
      <_headf>
           <nano>
             <_opf><fun>book</fun></_opf>
             <cterm>
                 <_opc><ctor>parts</ctor></_opc>
                 <var>title</var>
                 <var>author</var>
                 <var>contents</var>
                 <var>pages</var>
             </cterm>
          </nano>
      </_headf>
      <_foot>
          <var>author</var>
          <var>title</var>
      </_foot>
  </trans>
</rulebase>
```

Figure 3.37: RuleML XML representation of a transformation rule

Transformation rules can also be employed to produce an equivalent mode (e.g., an object model) from a given mode (e.g., relational model).

3.3.3. Rule Definition Meta-model (RDM)

Rule Definition Meta-model (RDM) is a MOF-based meta-model for Semantic Web Rule Language (SWRL). As SWRL includes Web Ontology Language (OWL) constructs, RDM represent an extension of Ontology Definition Meta-model (ODM) – a MOF-based meta-model for OWL. A proposal for meta-model for SWRL, with meta-model for OWL, is given in [Brockmans & Haase, 2006]. SWRL language has two concrete syntaxes, namely RDF/XML concrete syntax [Beckett, 2004], and OWL/XML concrete syntax [Hori et al., 2003].

3.3.3.1. Ontology Definition Meta-model (ODM)

Ontology Definition Metamodel [ODM, 2006] defines a meta-model for ontologies, by using the MOF meta-modeling language. ODM version that is used in this thesis is for OWL DL. It uses a notation that is accessible for users of UML, as well as for OWL DL ontology creators. We started from the ODM version that had been proposed in [Brockmans & Haase, 2006], and extended it and adopted it to be similar to ODM proposed in the OMG specification [ODM, 2006]. Figure 3.38 shows the main elements of ODM. Every element of an ontology is a `NamedElement`, and hence a member of an `Ontology`. Properties represent named binary associations in the modeled knowledge domain. OWL generally distinguishes three kinds of properties, so-called object properties, datatype properties and functional properties. A common generalization of them is given by the abstract meta-class `Property`. If a property is functional then its domain is a class. Object properties may additionally be inverse functional, transitive, symmetric, or inverse to another property. Their range is a class, while the range of datatype properties is a datatype. Properties can be related by using two types of axioms: property subsumption (`subPropertyOf`) specifies that the extension of a property is a subset of the related property, while property equivalence (`equivalentProperty`) defines extensional equivalence.

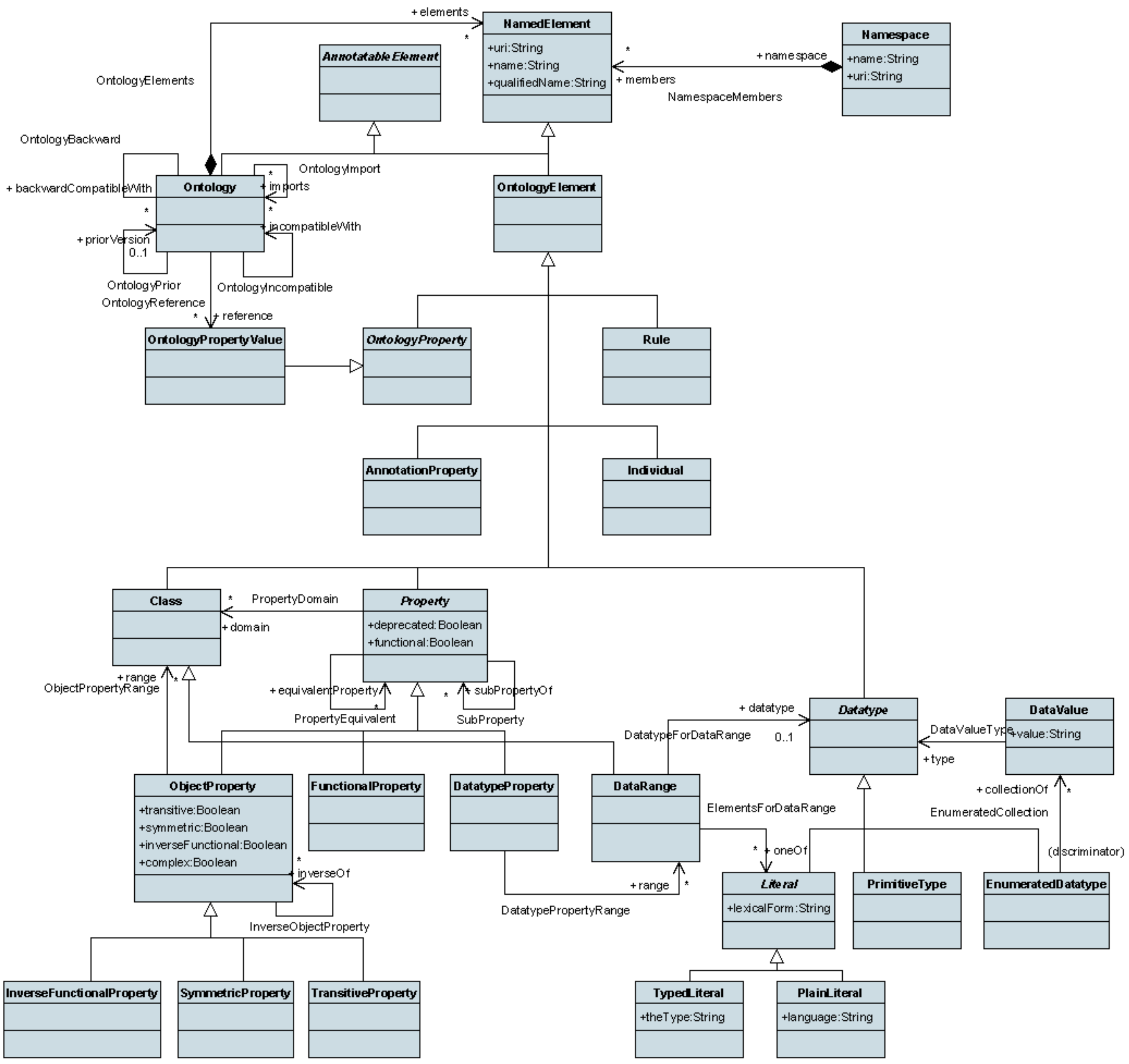


Figure 3.38: The main elements of Ontology Definition Meta-model (ODM)

The class hierarchy in ODM is shown in Figure 3.39. Class can be either a simple named class, or it can be built from the Boolean combination of classes, class restrictions, and enumerated classes. EnumeratedClass is defined through a direct enumeration of named individuals. Boolean combinations of classes are provided through ComplementClass, IntersectionClass and UnionClass.

The knowledge base elements (Figure 3.40) are part of an ontology. An Individual is an OntologyElement, that is, the subject of a PropertyValue. Naturally, an ObjectPropertyValue relates its subject with another Individual, whilst a DatatypePropertyValue relates its subject with a DataValue, which is an instance of DataType.

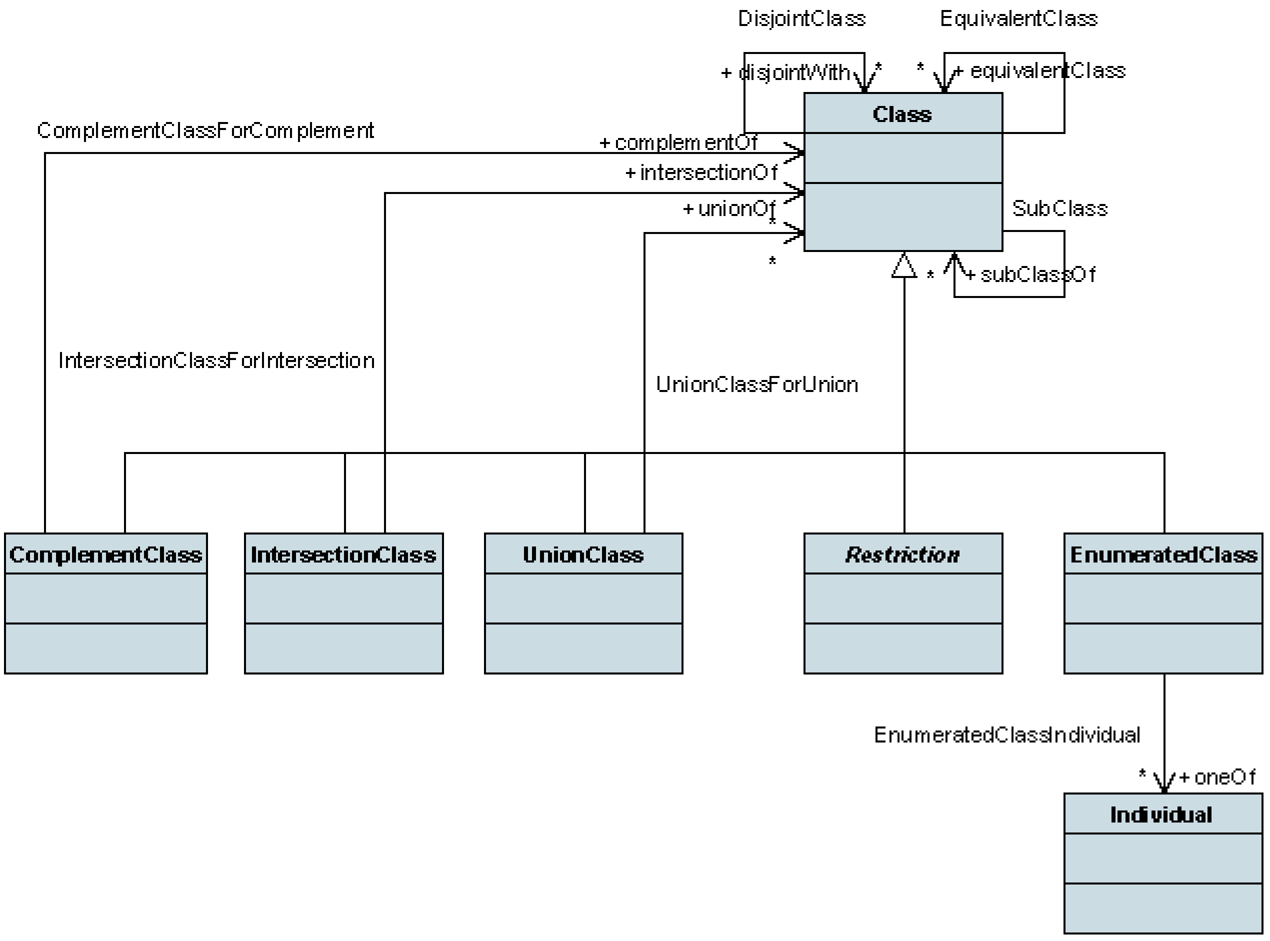


Figure 3.39: Classes in the ODM meta-model

Individuals can be related via three special axioms: the `sameAs` association allows users to state that two individuals are equivalent; the `differentFrom` association specifies that two individuals are not the same; `AllDifferent` is a simple notation for the pairwise difference of several individuals.

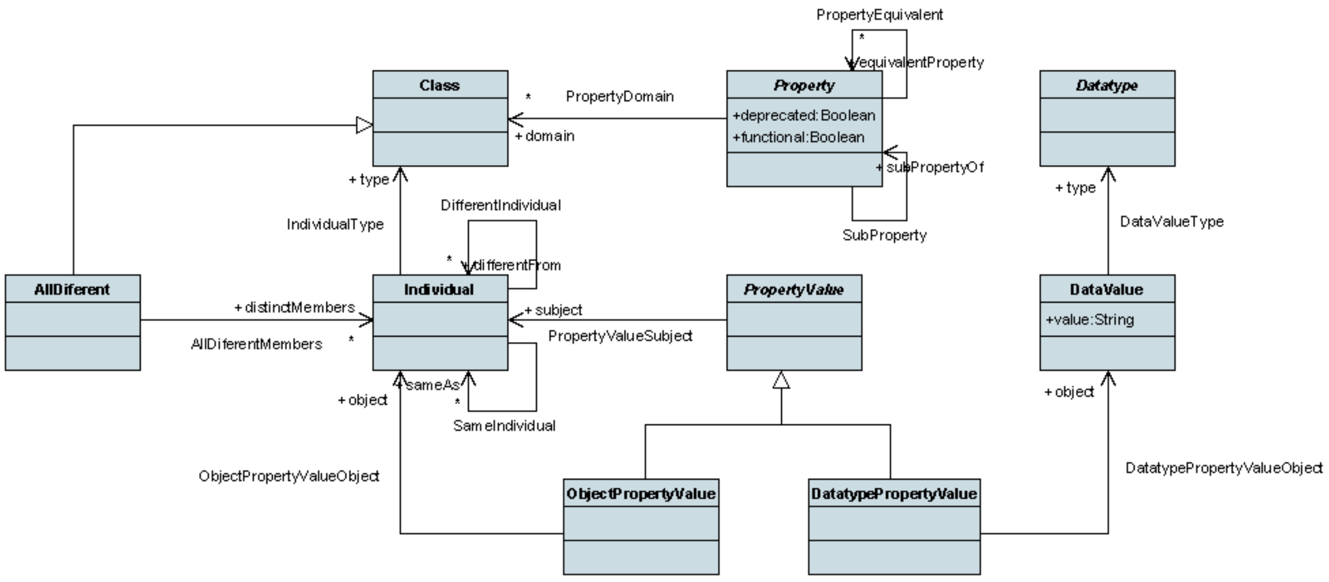


Figure 3.40: Knowledge base elements in the ODM meta-model

Special class type in ODM is the `Restriction` class (Figure 3.41). It represents restriction on some property, by using the `onProperty` association. Restrictions can be applied to all three types of properties. Restrictions can be on cardinalities: `CardinalityRestriction`, `MaxCardinalityRestriction` and `MinCardinalityRestriction`, or on values: `AllValuesFromRestriction` (class that have this restriction type take all values from a certain set), `SomeValuesFromRestriction` (class that have this restriction type take *at least one* value from a certain set) and `HasValueRestriction` (class that have this restriction type take always the same and only that value from a certain set).

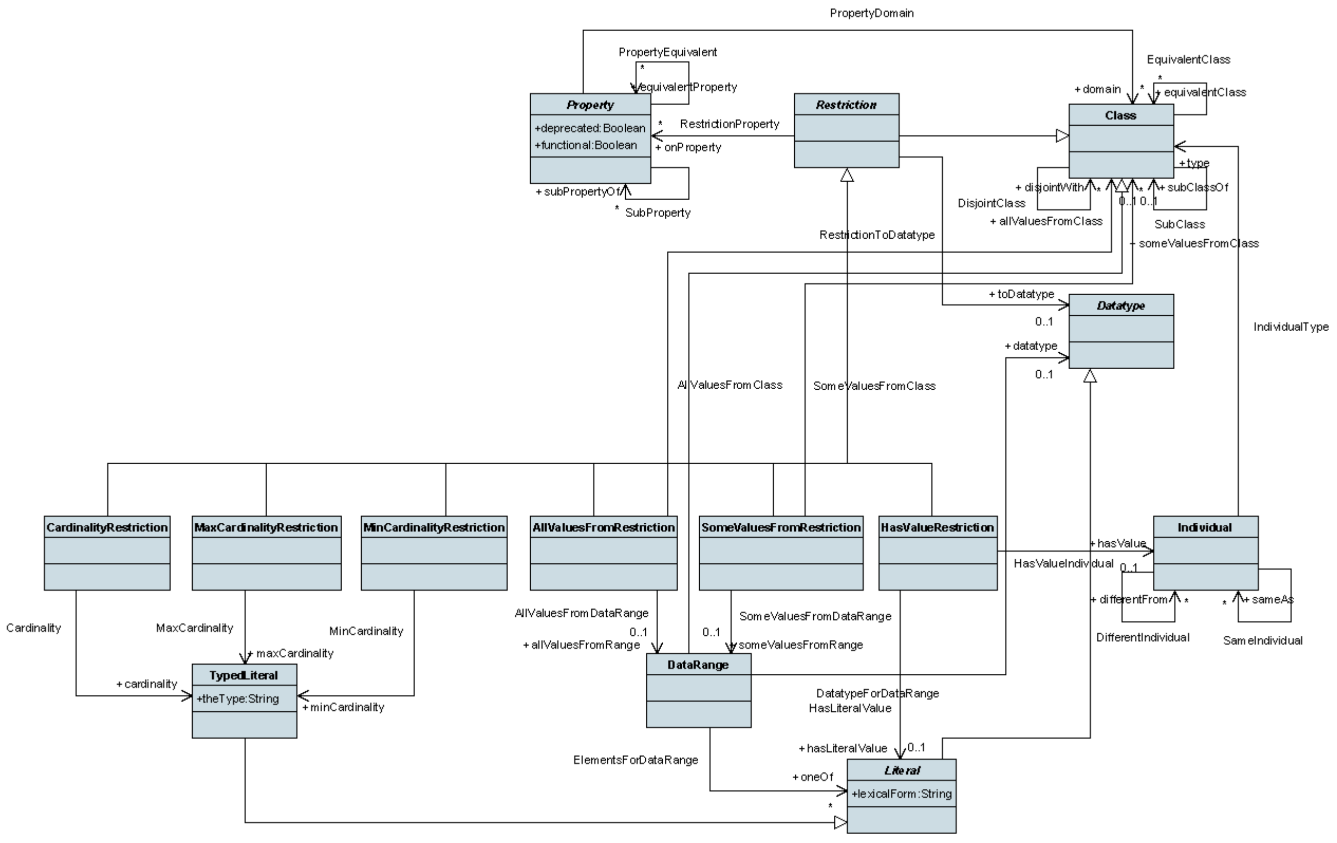


Figure 3.41: Restrictions in the ODM meta-model

3.3.3.2. Rules

SWRL defines rules as part of an ontology. The RDM meta-model defines the `Rule` class as a subclass of `OntologyElement` [Brockmans & Haase, 2006]. `OntologyElement` is defined in the ODM meta-model (Figure 3.39) as an element of `Ontology`, via the "elements" association between `NamedElement` and `Ontology`. As can also be seen in Figure 3.38, the class `OntologyElement` is a subclass of the class `AnnotatableElement`, which defines that rules can be annotated. For one ontology, it can be defined zero, one or multiple SWRL rules. An example of SWRL rule in XML-based concrete syntax is shown in Figure 3.42. It shows the use of built-ins for converting units of measure.

```
<ruleml:Implies xmlns:owlx="http://www.w3.org/2003/05/owl-xml"
                xmlns:swrlx="http://www.w3.org/2003/11/swrlx"
                xmlns:ruleml="http://www.w3.org/2003/11/ruleml">
      <owlx:Annotation>
            <owlx:Documentation>ex2:lengthInInches = ex1:lengthInFeet * 12
            </owlx:Documentation>
      </owlx:Annotation>
      <ruleml:body>
            <swrlx:datavaluedPropertyAtom swrlx:property="lengthInFeet">
                  <ruleml:var>instance</ruleml:var>
                  <ruleml:var>feet</ruleml:var>
            </swrlx:datavaluedPropertyAtom>
      </ruleml:body>
      <ruleml:head>
            <swrlx:builtinAtom swrlx:builtin="multiply">
                  <ruleml:var>inches</ruleml:var>
                  <ruleml:var>feet</ruleml:var>
                  <owlx:DataValue owlx:datatype="int">12</owlx:DataValue>
            </swrlx:builtinAtom>
            <swrlx:datavaluedPropertyAtom swrlx:property="lengthInInches">
                  <ruleml:var>instance</ruleml:var>
                  <ruleml:var>inches</ruleml:var>
            </swrlx:datavaluedPropertyAtom>
      </ruleml:head>
</ruleml:Implies>
```

Figure 3.42: A SWRL rule in XML-based concrete syntax [Horrocks et al., 2004]

A SWRL rule (`Rule` class) consists of an antecedent and a consequent, also referred to as the body and the head of a rule, respectively. Both the antecedent and the consequent consist of a set of atoms which can possibly be empty, as depicted by the multiplicity in Figure 3.43[29]. Every SWRL rule is an implication, which means that *if* all atoms of the antecedent hold, *then* the consequent holds. The same antecedent or consequent can be used in several rules, as indicated in the meta-model by the multiplicity of the association between the `Rule` class, on one hand, and the `Antecedent` class or the `Consequent` class on the other. The multiplicity in the opposite direction defines that the same atom can appear in several antecedents or consequents. According to the SWRL specification [Horrocks et al., 2004], every `Variable` that occurs in the `Consequent` of a rule must also occur in the `Antecedent` of that rule, a condition referred to as "safety". This constraint is defined as OCL invariant on class `Rule` (see class Rule in Figure 3.43).

[29] Remark: some associations are not shown in order to avoid the clutter.

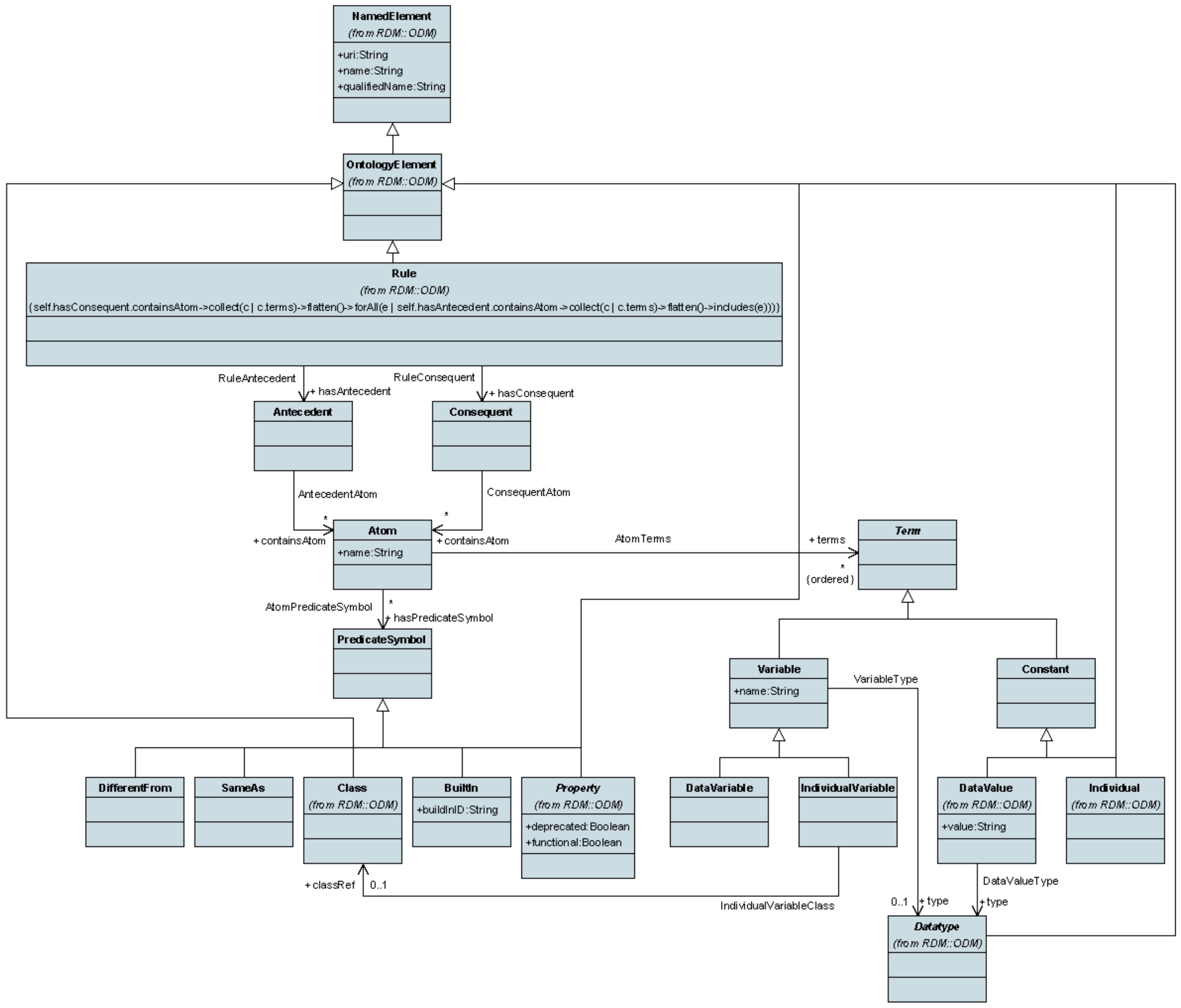


Figure 3.43: Rule Definiton Meta-model (adapted from [Brockmans & Haase, 2006])

3.3.3.3. Atoms, terms and predicate symbols

The atoms of the antecedent and the consequent of a rule consist of predicate symbols and terms. According to the SWRL specification [Horrocks et al., 2004], atoms can have different forms:

- `C(x)`, where `C` is an OWL description and `x` is an individual variable or an OWL individual; or `C` is an OWL data range and `x` is either a data variable or an OWL data value;
- `P(x, y)`, where `P` is an OWL individual valued property and `x` and `y` are both either an individual variable or an OWL individual; or `P` is an OWL datavalued property, `x` is either an individual variable or an OWL individual, and `y` is either a data variable or an OWL data value;
- `sameAs(x, y)`, where `x` and `y` are both either an OWL individual or an individual variable;
- `differentFrom(x, y)`, where `x` and `y` are both either an OWL individual or an individual variable;
- `builtIn(r, x, ...)`, where `r` is a built-in predicate and `x` is a data variable or an OWL data value. A `builtIn` atom could possibly have more than one variable or OWL data values.

The first of these, OWL description, data range and property, were already provided in ODM, namely as meta-classes `Class`, `DataRange` and `Property`, respectively. As can be seen in Figure 3.43, the predicates `Class`, `Property`, `SameAs`, `DifferentFrom` and `BuiltIn` are all defined as subclasses of the class `PredicateSymbol`, which is associated to `Atom`.

3.3.4. OCL meta-model

The OCL meta-model (i.e., abstract syntax for OCL version 2.0) is defined by using the MOF meta-modeling language [OCL, 2006]. In this abstract syntax, a number of meta-classes from the UML 2.0 meta-model are imported [UML, 2004]. These meta-classes are shown in the models with a transparent fill color. All meta-classes defined as part of the OCL abstract syntax are shown with a light gray background. The OCL meta-model is divided into several packages:

- The *Types* package describes the concepts that define the type system of OCL. It shows the types predefined in OCL as well as the types that are deduced from the UML models.
- The *Expressions* package describes the structure of OCL expressions.
- The *EnhancedOCL* package, the one that we have added to standard OCL meta-model in order to represent OCL constructs that are specific to OCL concrete syntax.

The OCL language has concrete syntax defined in EBNF technological space. However, the focus in this thesis is on OCL abstract syntax (i.e. meta-model). An example of an OCL invariant in concrete syntax is shown in Figure 3.44.

```
context Rental
  inv: self.additionalDriver.oclIsKindOf(QualifiedDriver)
```

Figure 3.44: OCL invariant in concrete syntax

3.3.4.1. The Types Package

Since OCL is a typed language, each expression has a type that is either explicitly declared or can be statically derived. Evaluation of the expression yields a value of this type. The model in Figure 3.45 shows the OCL types.

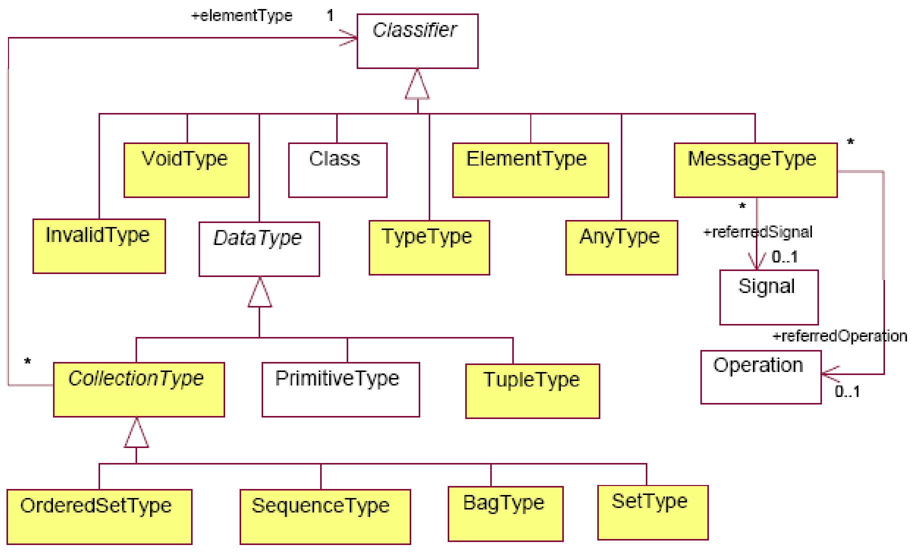


Figure 3.45: OCL types in the OCL meta-model

The basic type is the UML `Classifier`, which includes all subtypes of `Classifier` from the UML Superstructure [UML, 2005]. The OCL-specific types in OCL meta-model are:

- `CollectionType` - an abstract class that describes a list of elements of a particular given type. Its concrete subclasses are `SetType`, `SequenceType`, and `BagType`. An obligatory part of every collection type is the declaration of the type of its elements (i.e., a collection type is *parameterized* with an element type). In the meta-model, this is shown as an association from `CollectionType` to `Classifier` (see Figure 3.46).
- `TupleType` - combines different types into a single aggregate type. The parts of a `TupleType` are described by its attributes, each having a name and a type. There is no restriction on the kind of types that can be used as part of a tuple. In particular, a `TupleType` may contain other tuple types and collection types. Each attribute of a `TupleType` represents a single feature of a `TupleType`.
- `AnyType` - a special type that complies with all kinds of types except the collection types. `AnyType` has a unique instance named `OclAny`.
- `ElementType` - used for representing the type of the formal parameter of specific operations that need to refer to the elements of a model, such as the UML states in the pre-defined `oclInState` operation (this operation returns true if the object is in a certain condition).
- `MessageType` - describes OCL messages. Similar to the collection types, `MessageType` describes a set of types in the standard library.
- `VoidType` - represents a type that conforms to all types. The only instance of `VoidType` is `OclVoid`, which is further defined in the standard library.

3.3.4.2. The Expressions Package

This package defines the structures that OCL expressions can have. An overview of the inheritance relationships between all classes defined in this package is shown in Figure 3.46.

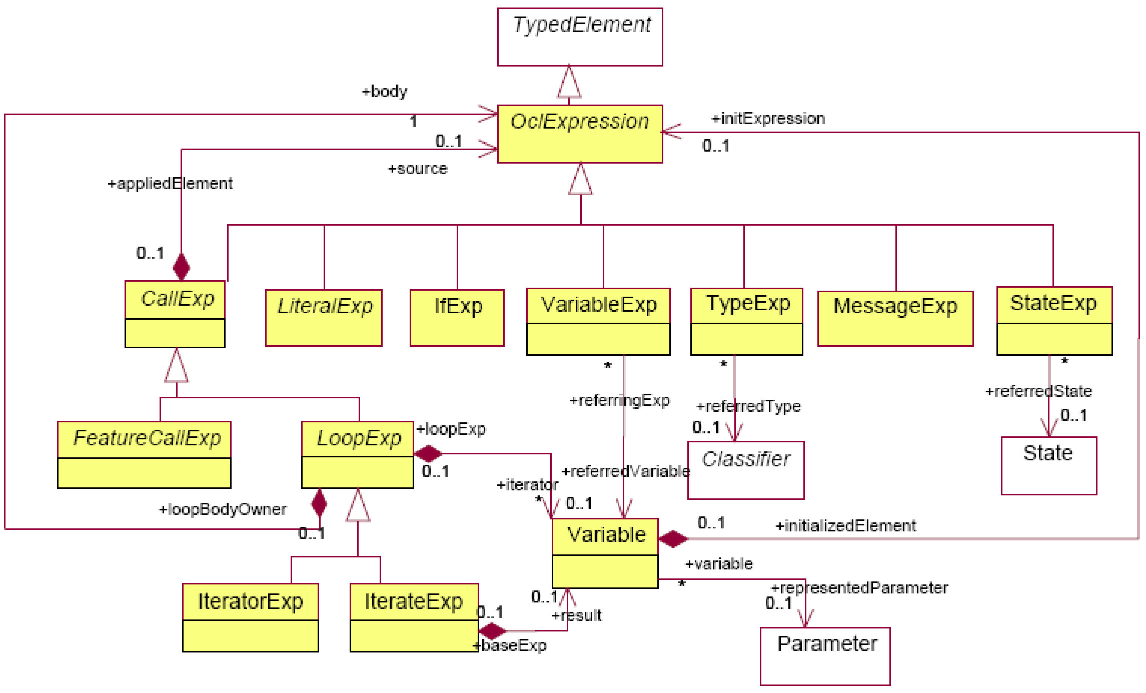


Figure 3.46: The basic structure of expressions in the OCL meta-model

The basic structure in the package consists of the classes:

- `OclExpression` - always has a type, which is usually not explicitly modeled, but derived.
- `CallExp` - has exactly one source, identified by an `OclExpression`.
- `FeatureCallExp` - generalizes all property calls that refer to `Features` in the UML meta-model. In Figure 3.47 various subtypes of `FeatureCallExp` are shown.
- `VariableExp` - an expression that consists of a reference to a variable. References to the variables *self* and *result* or to the variables defined by Let expressions are examples of such variable expressions.
- `LoopExp` - an expression that represents a loop construct over a collection. It has an iterator variable that represents the elements of the collection during iteration. The body expression is evaluated for each element in the collection.
- `IterateExp` - an expression that evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. An iterate expression evaluates its *body* expression for each element of its *source* collection. The evaluated value of the *body* expression in each iteration-step becomes the new value for the *result* variable for the succeeding iteration-step. The result can be of any type and is defined by the *result* association.
- `IteratorExp` - is an expression that evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its *source* collection and results in a value. The type of the iterator expression depends on the name of the expression, and sometimes on the type of the associated *source* expression. It represents all other predefined collection operations that use an iterator. This includes *select*, *collect*, *reject*, *forAll*, *one*, *exists*, etc.
- `LiteralExp` - an expression with no arguments that produces a value. In general, the result value is identical with the expression symbol. This includes things like the integer *1* or literal strings like '*this is a string*'.
- `StateExp` - an expression used for refering to the state of a class within an expression. It is used to pass directly to the pre-defined operation *oclIsInState* the reference of a state of a class defined in the UML model.
- `TypeExp` - an expression used for refering to an existing meta type within an expression. In particular, it is used to pass the reference of the meta type when invoking the operations *oclIsKindOf*, *oclIsTypeOf* and *oclAsType*.

`FeatureCallExp` can refer to any subtype of *Feature*, as defined in the UML kernel (Figure 3.47).

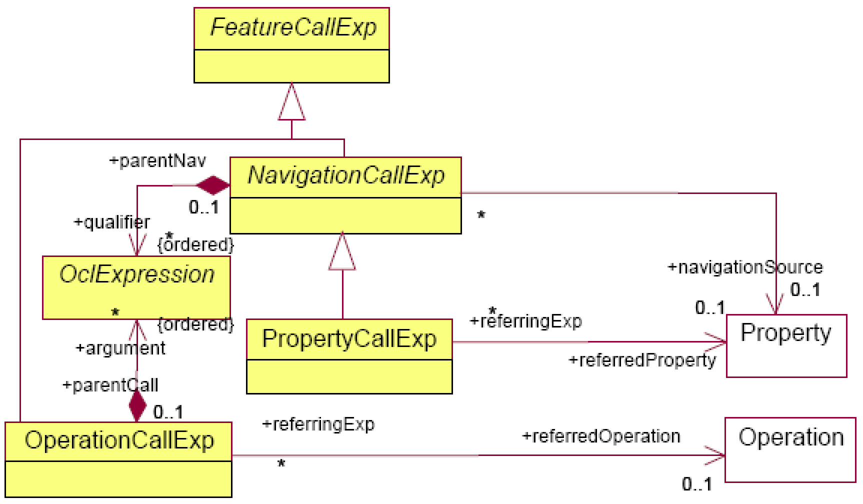


Figure 3.47: `FeatureCallExp` in the *Expressions* package of the OCL meta-model

- `PropertyCallExp` - a reference to an `Attribute` of a `Classifier` defined in a UML model. It evaluates to the value of the attribute.
- `OperationCallExp` - refers to an operation defined in a `Classifier`. This expression may contain a list of argument expressions, if the operation is defined to have parameters. In this case, the number and types of the arguments must match the parameters.

`IfExp` results in one of two alternative expressions depending on the evaluated value of the *condition* (see Figure 3.48). Both *thenExpression* and *elseExpression* are mandatory. The rationale is that an expression should always result in a value, which cannot be guaranteed if the else part is left out.

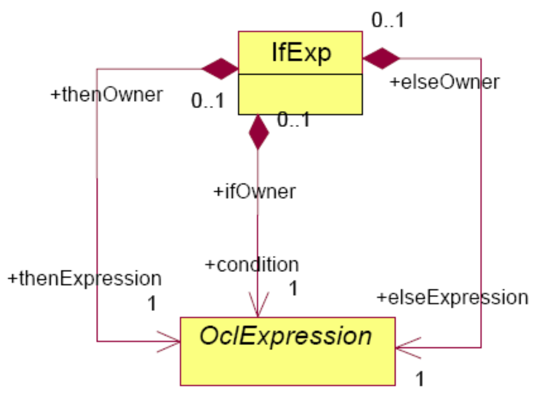


Figure 3.48: If expression in the *Expressions* package of the OCL meta-model

Figure 3.49 shows a hierarchy of literal expressions in the OCL meta-model. It also presents enumeration types and enumeration literals.

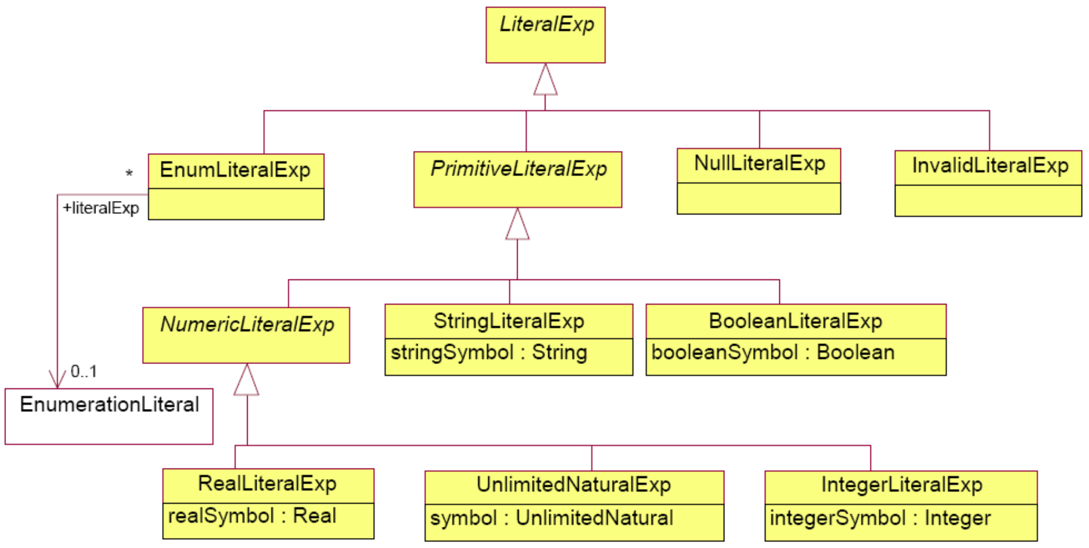


Figure 3.49: Literal expression in the *Expressions* package of the OCL meta-model

The classes of literal expressions in the OCL meta-model are:

- `BooleanLiteralExp` - represents the values (true or false) of the predefined type `Boolean`.
- `CollectionItem` - represents an individual element of a collection.
- `CollectionKind` - an enumeration of the kinds of collections. Possible values are *Set*, *OrderedSet*, *Bag*, and *Sequence*.
- `CollectionLiteralExp` - represents a reference to a collection literal (Figure 3.50).
- `CollectionLiteralPart` - a member of the collection literal.
- `CollectionRange` - represents a range of integers.
- `EnumLiteralExp` - represents a reference to an enumeration literal.
- `IntegerLiteralExp` - denotes a value of the predefined type `Integer`.
- `NumericLiteralExp` - denotes a value of either the type `Integer` or the type `Real`.
- `PrimitiveLiteralExp` - literal denotes a value of a primitive type.
- `RealLiteralExp` - denotes a value of the predefined type `Real`.
- `StringLiteralExp` - denotes a value of the predefined type `String`.
- `TupleLiteralExp` - denotes a tuple value. It contains a name and a value for each part of the tuple type.

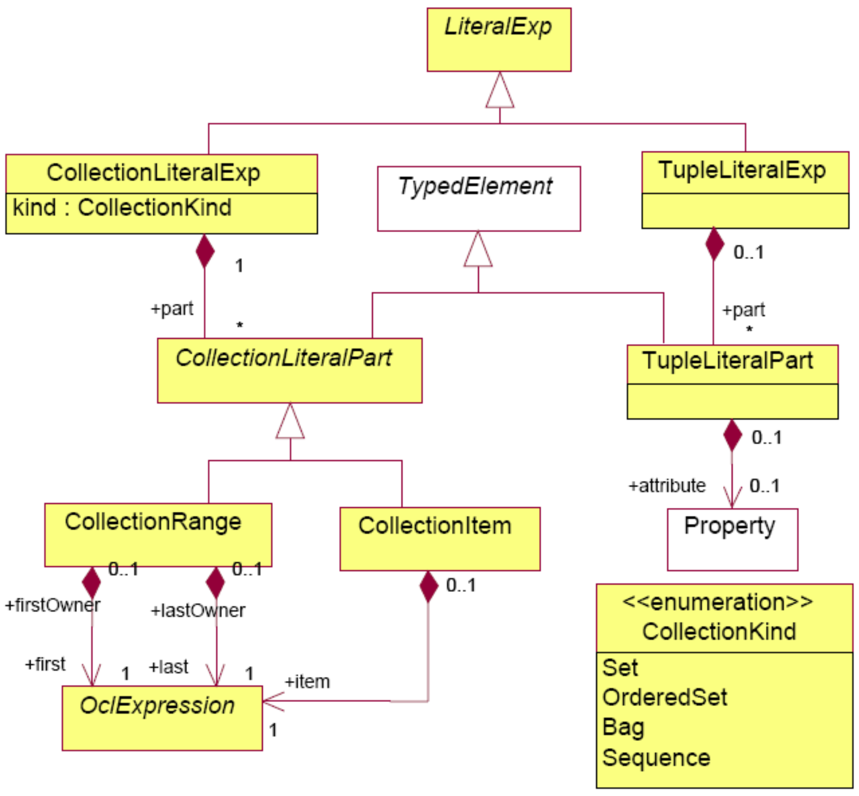


Figure 3.50: Collections in the *Expressions* package of the OCL meta-model

`LetExp` is a special expression that defines a new variable with an initial value. A variable defined by `LetExp` cannot change its value – this value is always evaluated from the initial expression. The variable is visible in the *in* expression (Figure 3.51).

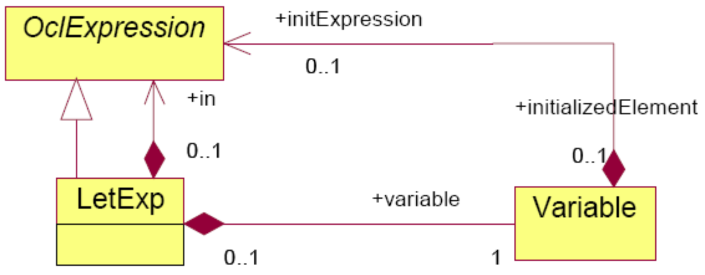


Figure 3.51: Let expression in the *Expressions* package of the OCL meta-model

3.3.4.3. The EnhancedOCL package

Since the standard specification of the OCL meta-model [OCL, 2006] does not contain support for OCL invariants, in this thesis the *EnhancedOCL* package is defined in order to fill this gap. This package contains the `Invariant` class, as a subclass of the `OclModuleElement` class (see Figure 3.52). `OclModuleElement` represents a superclass for:

- OCL invariant elements (the `Invariant` class);
- OCL operations and properties, i.e., "def" elements (the abstract class `OclFeature`) that are represented with classes `OclOperation` and `OclProperty`, respectively;
- OCL derivation rules (the `DeriveOclModuleElement` class).

`OclModuleElement` contains invariant context definition that is represented with the `OclContextDefinition` class. In addition, the `OclModule` class is introduced to represent a basic class in OCL model, and it contains other `OclModuleElement`'s.

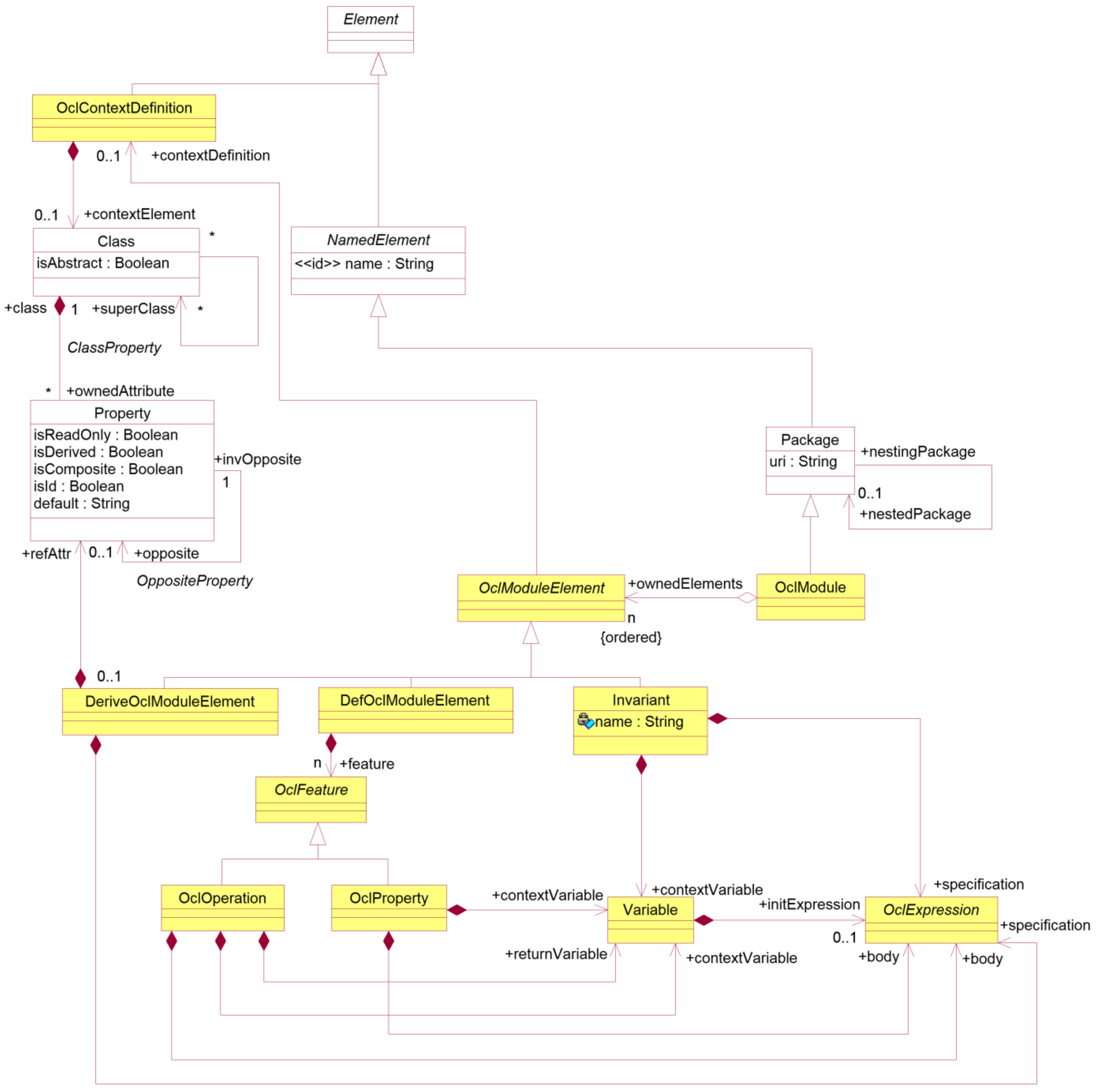


Figure 3.52: Elements of the *EnhancedOCL* package in the OCL meta-model

4. TRANSFORMATIONS OF WEB RULES - CONCEPTUAL SOLUTION

This section describes conceptual solution for mappings between rule language elements, as well as transformations between concrete models in those languages. The presented solution uses R2ML meta-model as central meta-model for sharing rules between OWL/SWRL and UML/OCL languages.

4.1. R2ML AS CENTRAL RULE MARKUP LANGUAGE

Rule Interchange Format (RIF) [Ginsberg, 2006] is a W3C initiative that should define an intermediary language between various rule languages, without providing a formally defined semantic foundation for reasoning on the Web such as OWL for ontologies. Currently, this initiative defines a set of requirements and use cases for sharing rules on the Web. However, there is no official submission to this initiative yet.

RuleML is a language that tends to be a future proposal for RIF. RuleML is a markup language for publishing and sharing rule bases on the World Wide Web [Hirtle et al., 2006]. RuleML builds a hierarchy of rule sublanguages upon the XML, RDF, XSLT, and OWL, and it is based on the Datalog. The current RuleML hierarchy consists of derivation (e.g., SWRL, FOL) and production rules (e.g., Jess). RuleML rules are defined in the form of an implication between an antecedent and consequent, meaning that whenever the logical expression in the antecedent holds, then the consequent must also hold. However, an important constraint of RuleML is that it cannot fully represent all the constructs of various languages such as OCL or SWRL and it cannot satisfy many RIF requirements.

In contrast with RuleML, there is REWERSE I1 Rule Markup Language (R2ML) as a language that covers all RIF's requirements for sharing rules, and hence solves some of the RuleML constraints. Because of that, R2ML is used in this thesis, as the central rule markup language for sharing rules between different rule languages.

The complete scenario for transformating rules between different rule languages, by using R2ML as central language, is shown in Figure 4.1

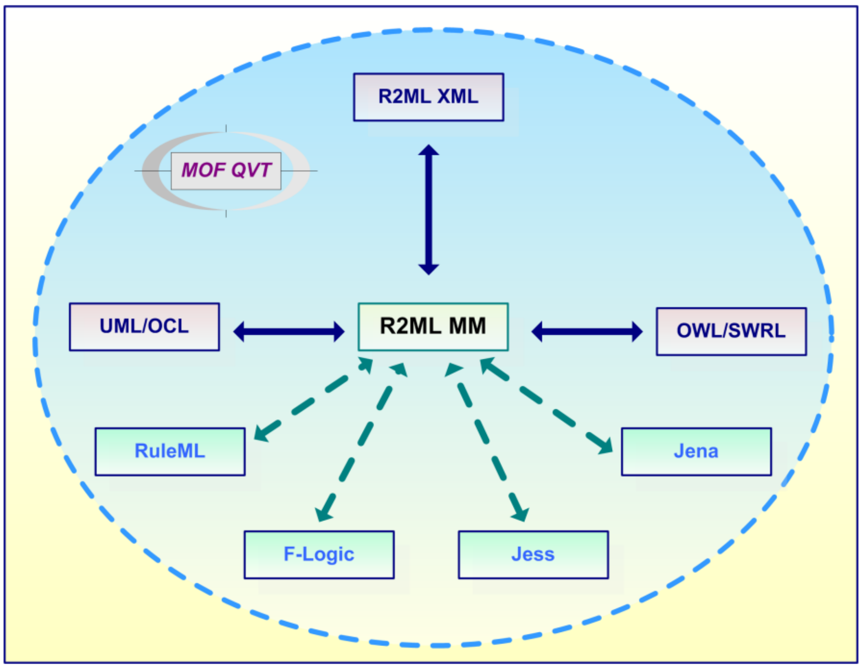


Figure 4.1: Complete transformations scenario between SWRL, OCL and other Rule languages via R2ML

In this transformation scenario, we show implementation of rule sharing on the Web between SWRL language and OCL language, via R2ML language, while mappings with other rule languages like: RuleML, F-Logic, Jess, and Jena are also possible. The SWRL and OCL use different concrete syntaxes, defined in the XML and EBNF technological spaces, respectively. Thus, defining and implementing mappings between languages is done on the level of their abstract syntax (i.e., meta-models in MOF technological space), as this actually allows focusing on mappings between language constructs, rather than on the implementation details of their concrete syntaxes. This means that the language definitions are done by using abstract syntax, while concrete (visual or textual) syntaxes are employed to physically represent rules. As R2ML has an XML-based concrete syntax, we defined and implemented mappings between its abstract and concrete syntaxes. Figure 4.2 shows implemented transformations between concrete and abstract syntaxes (i.e., meta-models), of different rule markup languages, as well as transformation principle between their abstract syntaxes.

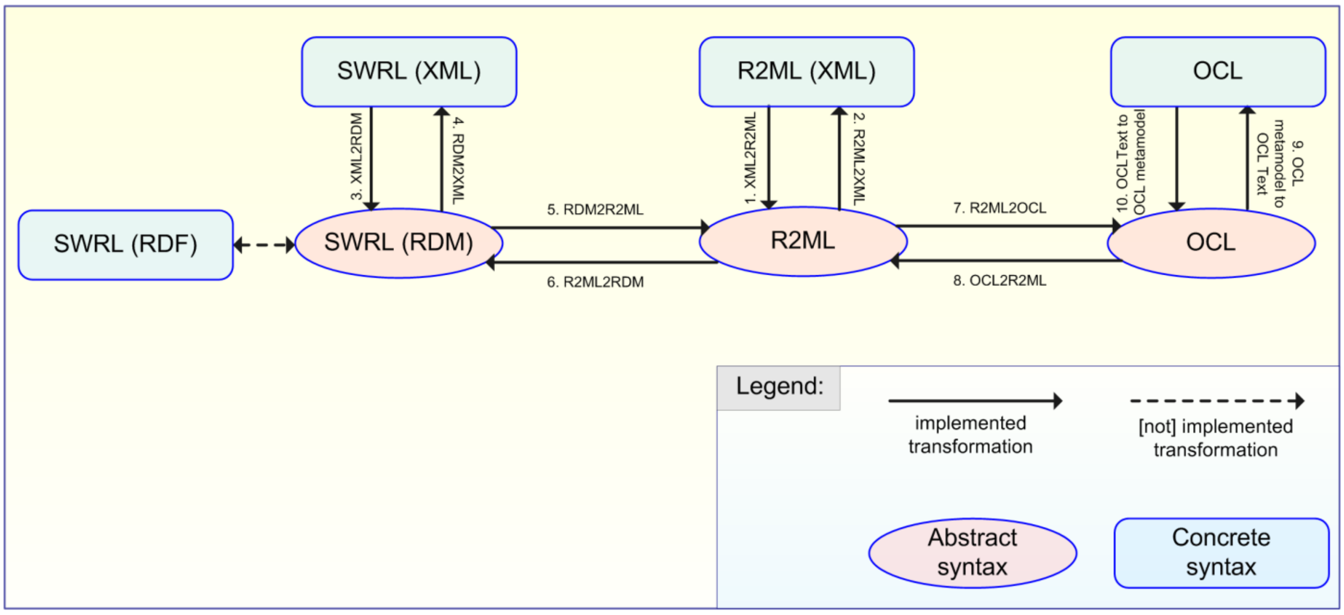


Figure 4.2: Implemented ATL transformations between concrete and abstract syntaxes of different rule languages

Because the transformations between the SWRL, R2ML and OCL abstract syntaxes (in Figure 4.2, marked with arrows) are implemented in the MOF technological space, rules from the XML technological space (for SWRL) and the EBNF technological space (for OCL), are firstly converted to the MOF technological space. The conversion of rules presented in the XML technological space to the MOF technological space, is done by using the XML injector (i.e., XML extractor). This operation has an extremely "low cost", because we use XML injector and extractor, which are distributed as a tool along with the ATL engine. XML injector takes an XML file as the input and produces its equivalent model that is conformant to the XML meta-model defined in MOF, while XML extractor produces an XML file from model conformant to the meta-model defined in either ECore or a MOF. For conversion between EBNF and MOF technological spaces, we used the EBNF injector/extractor, that works similar as the XML injector/extractor, but for rules defined in the EBNF technological space, i.e., for OCL rules. When rules from different technological spaces are brought to the MOF technological space, then we can use some QVT-based tool for executing transformations on them. Rules represented as instances of the SWRL language meta-model (RDM meta-model), i.e., OCL language (OCL meta-model), transform to the R2ML model, and after that from the R2ML model to the some target model. A rule that is represented as an instance of the R2ML meta-model (in MOF TS) can be serialized in R2ML

XML format (XML TS), and loaded from R2ML XML format to the instance of the R2ML meta-model.

4.2. TRANSFORMATIONS BETWEEN SWRL AND R2ML: BRIDGING XML AND MOF TECHNOLOGICAL SPACES

Transformations between concrete and abstract syntaxes of rule languages, represented in different technological spaces (as XML for concrete and MOF for abstract syntaxes), are done in two directions. First, we have rule transformations from the XML technological space to meta-model instance in MOF technological space, and second, transformation of meta-model instance from the MOF technological space to the rule in concrete syntax in XML the technological space.

4.2.1. Transformations between the R2ML XML Schema and the R2ML model

The fact that concrete syntaxes of SWRL and R2ML are defined in the XML technological space, and their abstract syntaxes (i.e., meta-models) are defined in MOF the technological space, emphasizes the necessity to bring concrete syntax elements of these languages to their abstract syntax elements. For this translation we used the XML meta-model (based on [XML, 2006]) as a bridge between the XML and MOF technological spaces. XML meta-model (shown in Figure 4.3) describes XML document that consists of one root node.

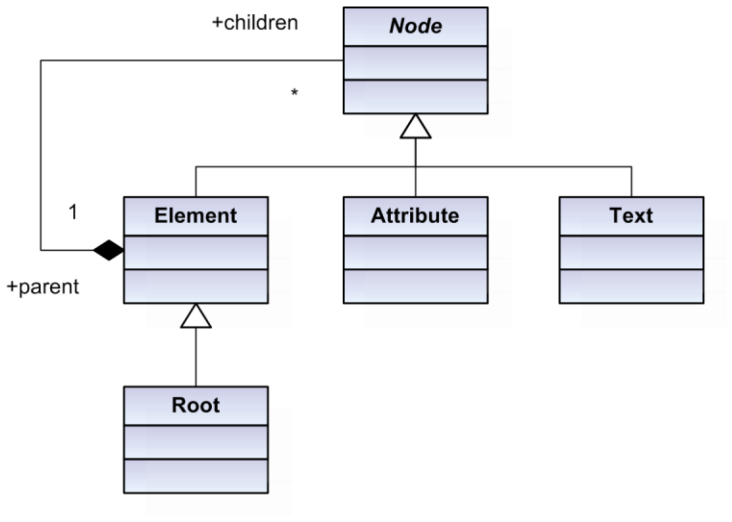


Figure 4.3: XML meta-model [XML, 2006]

`Node` is an abstract class that has three children: `Element`, `Attribute`, and `Text`. `Element` represents tags, e.g., tag with name xml: `<xml></xml>`. `Element` can contain multiple nodes. `Attribute` represents attributes that can be defined in some tag, e.g., attr attribute: `<xml attr="attribute value"/>`. `Text` is special node that does not look like tag but represents a set of characters.

In order to transform some XML document from the XML technological space to the XML model (instance of XML meta-model) of that same rule in the MOF technological space, we used XML injector class as part of ADT toolkit (ATLAS Development Tools [ATL, 2007]). XML injector is one of several available injectors in ADT environment, and it is available as Eclipse plug-in. XML injector is implemented as `XMLInjector` Java class, that uses SAX Parser[30] which takes a XML file as input

[30] http://www.saxproject.org

and returns created XML model, i.e., root element. XML model that is generated with `XMLInjector` class can be serialized to file, in ECore or MOF XMI format.

In the same way, by using built-in the `XMLExtractor` class, we can get XML document in the XML technological space from a XML model (MOF TS).

Figure 4.4. shows the transformation scenario of mapping between the R2ML XML concrete syntax and R2ML abstract syntax (i.e., meta-model). From this figure we can see that it is possible to do transformations in both directions. Details about these transformations are shown in the following two sections.

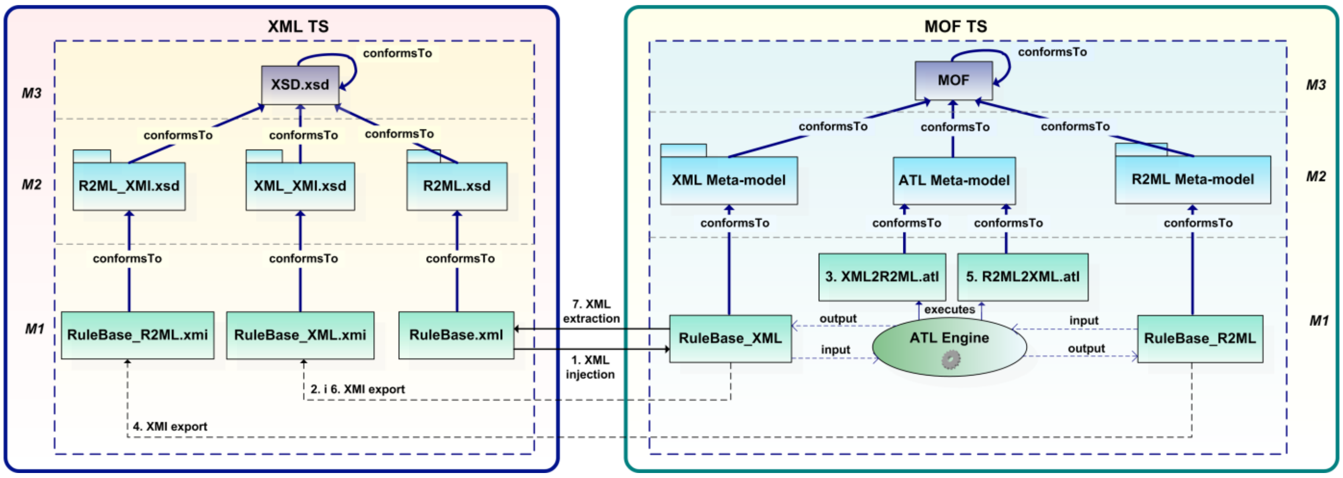


Figure 4.4: The transformation scenario: R2ML XML format into the R2ML meta-model and vice versa

4.2.1.1. Transforming the R2ML XML Schema into the R2ML model

Transformation of R2ML rule from R2ML XML concrete syntax into the R2ML abstract syntax (i.e., meta-model) is transformation 1. XML2R2ML in Figure 4.2. The transformation process of R2ML XML Schema into the R2ML meta-model consists of two primary steps as follows.

Step 1. XML injection from the XML technical space to the MOF technical space. This means that we have to represent R2ML XML documents (RuleBase.xml from Figure 4.3) into the form compliant to the MOF. We use the XML injector that transforms R2ML XML documents (written w.r.t. the R2ML XML Schema, R2ML.xsd from Figure 4.4) into the models conforming to the MOF-based XML metamodel (step 1 in Figure 4.4).

An XML model (RuleBase_XML in Figure 4.4), created by the XML injector, is located in the M1 layer of the MDA. This means that the XML injector instantiates the MOF-based XML meta-model. We can manipulate with these models like with any other type of MOF-based models. Thus, such XML models can be represented in the XMI format (step 2 in Figure 4.4) and this XMI format can be regarded as an implicitly defined XML schema (XML_XMI.xsd) compliant to the XML meta-model. Since ATL is based on MDR, we can employ MDR’s tool for exporting XMI documents (e.g., RuleBase_XML.xmi). For example, an alethic integrity rule from Figure 3.4 is represented in the XMI document in Figure 4.5.

```
<XML.Element xmi.id = 'a1' name = 'r2ml:AlethicIntegrityRule'  value = ''>
 <XML.Element.children>
   <XML.Attribute xmi.id = 'a2' name = 'r2ml:id' value = 'IR001'/>
   <XML.Element xmi.id = 'a3' name = 'r2ml:constraint' value = ''>
     <XML.Element.children>
       <XML.Element xmi.id = 'a4'
        name = 'r2ml:UniversallyQuantifiedFormula'
        value = ''>
         <XML.Element.children>
           <XML.Element xmi.id = 'a24' name = 'r2ml:ObjectVariable' value = ''>
              <XML.Element.children>
                 <XML.Attribute xmi.id = 'a12'
                  name = 'r2ml:name' value = 'r1'/>
                 <XML.Attribute xmi.id = 'a13'
                  name = 'r2ml:classID' value = 'Rental'/>
              </XML.Element.children>
           </XML.Element>
           <!--..-->
         </XML.Element.children>
         <!--..-->
        </XML.Element>
        <!--..-->
      </XML.Element.children>
      <!--..-->
    </XML.Element>
  </XML.Element.children>
</XML.Element>
```

Figure 4.5: The integrity rule from Figure 3.4 represented in the XML XMI format

Step 2. A transformation of XML models into R2ML models. We transform an XML model (RuleBase_XML) created in Step 1 into an R2ML model (RuleBase_R2ML) by using an ATL transformation named XML2R2ML.atl (step 3 in Figure 4.4). The output R2ML model (RuleBase_R2ML) conforms to the R2ML meta-model. In the XML2R2ML.atl transformation, source elements from the XML model are transformed into target elements of the R2ML model. The XML2R2ML.atl transformation is done on the M1 level (i.e., the model level) of the MDA. This transformation uses information about elements from the M2 (meta-model) level, i.e., meta-models defined on the M2 level (i.e., the XML and R2ML meta-models) in order to provide transformations of models on the level M1. It is important to point out that M1 models (both source and target ones) must be conformant to the M2 meta-models.

Table 4.1 shows an excerpt of mappings between the R2ML XML Schema, XML meta-model, and R2ML meta-model. For XML Schema complex types, an instance of the XML meta-model element is created through the XML injection described in Step 1 above. Such an XML element is then transformed into an instance of the R2ML meta-model element by using the XML2R2ML.atl transformation (Step 2). The ATL transformation is done for classes, attributes, and references.

Table 4.1: An excerpt of mappings between the R2ML XML schema and the R2ML meta-model

R2ML XML Schema	XML meta-model	R2ML meta-model	Description

`RuleBase`	`Root`
	`name =`
	`'r2ml:RuleBase'`
		`RuleBase`	Captures a collection of rules.

`IntegrityRuleSet`	`Element`
	`name =`
	`'r2ml:IntegrityRuleSet'`
		`IntegrityRuleSet`	Captures a set of integrity rules.

`AlethicIntegrityRule`	`Element`
	`name =`
	`'r2ml:AlethicIntegrityR`
	`ule'`
		`AlethicIntegrityRule`	Represents an alethic integrity rule.

`ObjectVariable`	`Element`
	`name =`
	`'r2ml:ObjectVariable'`
		`basContVoc.`
		`ObjectVariable`
			Represents an object variable.

`ObjectVariable`
 	
`attribute`
`ref="r2ml:name"`
	`Attribute`
	`name = 'r2ml:name'`
	`value = 'r1'`
		`basContVoc.`
		`ObjectVariable`
		
		`attribute name =`
		`'r1'`
			Represents the name attribute of an object variable.

Mappings between elements of the XML meta-model and elements of the R2ML meta-model are defined as a sequence of rules in the ATL language. These rules use additional helpers in defining mappings (similar to functions in programming languages). Each rule in the ATL has one input element (i.e., an instance of a meta-class from a MOF based meta-model) and one or more output elements. In fact, the ATL transformation takes an input XML model from a model repository and creates a new model compliant to the R2ML meta-model. This actually means that we instantiate the R2ML meta-model (M2 level), i.e., create R2ML models (M1 level). In the XML2R2ML.atl transformation, we mainly use ATL matched rules. A matched rule matches a given type of source model element, and generates one or more kinds of target model elements. These rules are activated by the ATL rule engine for each element of a given type of the source model.

In development of the XML2R2ML.atl transformation, we have created a library of helper operations (called XMLHelpers.atl), that contains all operations necessary for work with the XML meta-model, and which may use in any transformation of type *XML2SomeDomainMetamodel*.

Figure 4.6 gives an example of a matched rule which is in fact an excerpt of the X2ML2R2ML.atl transformation for the `Root` class (`XML!Root`) of the XML meta-model. The ATL rule `RuleBase` transforms the `Root` of the XML model (RuleBase_XML) into the `RuleBase` element (i.e., `R2ML!RuleBase` in Figure 4.6) of the R2ML meta-model. The `RuleBase` element (the `Root` of an XML tree) captures a collection of different `RuleSet`-s (i.e., Derivation, Integrity, Reaction or Production). Each `RuleSet` includes rules of their own kind (e.g., `IntegrityRuleSet` includes only `IntegrityRule`-s). From the ATL rule shown in Figure 4.6, we invoke rules for nested elements (e.g., `DerivationRuleSet` and `IntegrityRuleSet`) as well as for the `ruleBaseID` attribute.

```
module XML2R2ML;
create OUT : R2ML from IN : XML;

rule RuleBase {
  from
     i : XML!Root
  to o : R2ML!RuleBase (
    ruleBaseID <- i.getAttrVal('xmlns:r2ml'),
         rules <- XML!Element.allInstances()->select(e | e.name = 'r2ml:DerivationRuleSet'
                                                       or e.name = 'r2ml:IntegrityRuleSet')
    )
```

Figure 4.6: ATL matched rule for `RuleBase` R2ML element

However, this type of rules is not suitable for all input elements of the XML meta-model that are transformed into the R2ML meta-model. This is typical for a situation when several input elements represent the same entity, while in the target model we have to have only one definition of that element and others should only refer to that one. For example, the `ObjectVariable` XML element in the input rules might be used several times and the only way to know that all of them refer to the same variable is by the value of its name attribute (e.g., customer). However, in the target R2ML meta-model, an `ObjectVariable` should only be defined once, while all other parts of the model can only refer to that definition. Figure 3.4 shows an R2ML `IntegrityRule`, while Figure 4.7 gives the same rule in a form of a UML object diagram representing instances of the R2ML MOF-based meta-model. In Figure 4.7, we can see that the *r1* `ObjectVariable` element is a unique for the whole rule, and it appears at 3 different places in the rule. This means that an XML tree contains three different nodes referring to the same object, i.e., the *r1* `ObjectVariable`. On the other hand, the instance of the R2ML MOF-based meta-model contains one and only one node referring to that object (`r1:ObjectVariable`), as it is shown in Figure 4.7. Three links to that object represent three different appearances of that `ObjectVariable` in the whole rule.

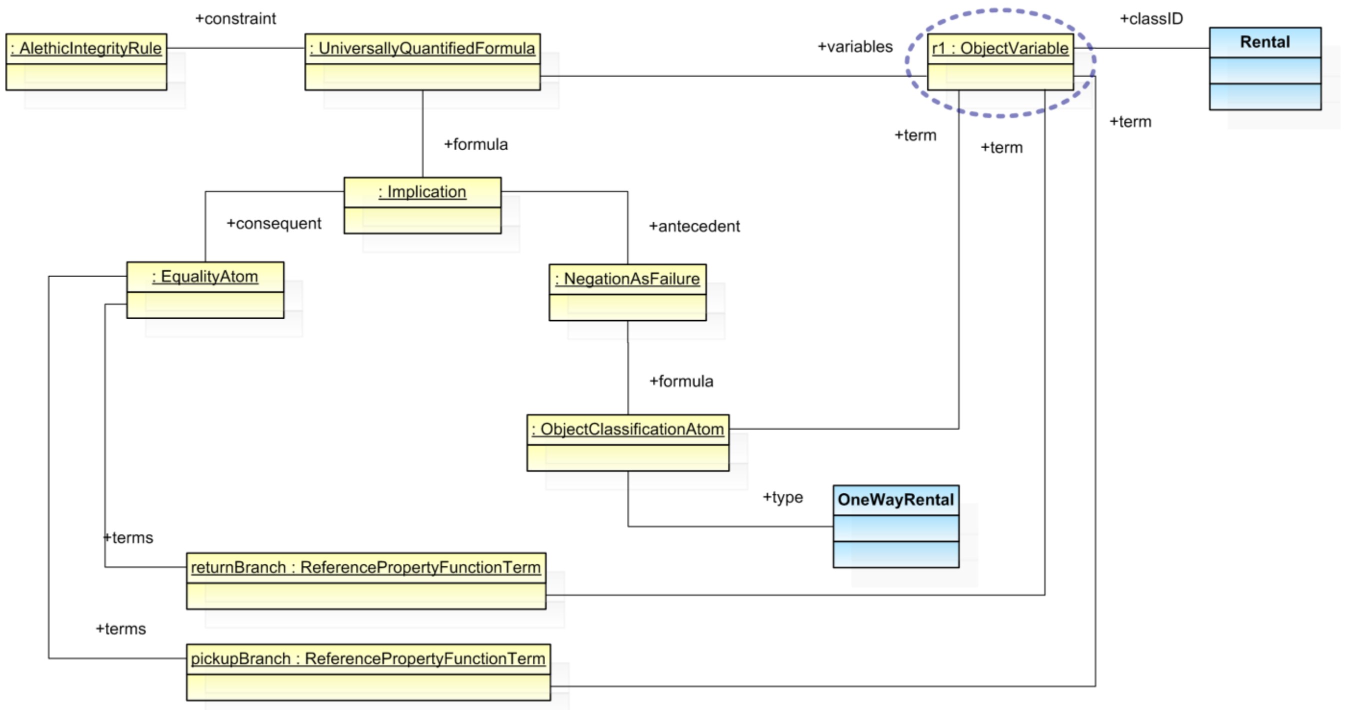


Figure 4.7: A UML object diagram that represents the integrity rule shown in Figure 3.4 in the R2ML XML format

It is obvious that matched rules are unsuitable for this type of transformation. As a solution to this issue, ATL's "unique lazy" rules are employed. They are "lazy", since they invoke explicitly from some other rule. They are "unique", since they only create an element of the target model once for a specific element of the source model, while all other executions of the same rule for that specific input element create references to the only target definition. This type of rules substitutes called templates in XSLT with the main distinction that XSLT does not posses any mechanism for unique element definitions, and is very hard to implement in XSLT [Jovanović & Gašević, 2005]. The unique lazy ATL rule that transforms `ObjectVariable`-s from the XML meta-model into the R2ML meta-model is shown in Figure 4.8. It transforms an Element of the XML meta-model (`XML!Element`) whose attribute name has the value "`r2ml:ObjectVariable`". This element is transformed into the

ObjectVariable element of the R2ML meta-model (R2ML!ObjectVariable) following all conditions given above.

```
unique lazy rule ObjectVariable {
      from i : XML!Element (
                  i.name = 'r2ml:ObjectVariable'
            )
      to
            ov : R2ML!ObjectVariable (
                  classRef <- i.children->select(c | c.oclIsKindOf(XML!Attribute) and
                              c.name = 'r2ml:classID')->collect(e | thisModule.ClassRule(e))->first(),
                  name <- i.getAttrVal('r2ml:name'),
                  typeCategory <- #individual
            )
}
```

Figure 4.8: A unique lazy rule of ObjectVariable element, in the transformation from the R2ML XML format to the R2ML meta-model

After applying the above ATL rules to the input XML models, R2ML models (RuleBase_R2ML) are stored in the model repository. Such R2ML models can be exported in the form of the R2ML XMI documents (e.g., RuleBase_R2ML.xmi in Figure 4.3), by using NetBeans Metadata Repository (MDR) functionalities. For example, for rules shown in Figure 4.5 (i.e., in the XMI format of the XML meta-model), R2ML XMI document in Figure 4.9 is attained. Figure 4.9 shows a RuleBase that contains IntegrityRulesSet with one AlethicIntegrityRule. AlethicIntegrityRule then constraints UniversallyQuantifiedFormula, which contains variable declarations.

```
<R2ML.RuleBase xmi.id="a1" ruleBaseID="OCL">
   <R2ML.RuleBase.rules>
      <R2ML.IntegrityRuleSet xmi.id="a2">
         <R2ML.IntegrityRuleSet.rules>
            <R2ML.Rules.AlethicIntegrityRule xmi.id="a3" ruleID="">
               <R2ML.Rules.IntegrityRule.constraint>
                  <R2ML.Formulas.UniversallyQuantifiedFormula xmi.id="a7">
                      <R2ML.Formulas.QuantifiedFormula.variables>
                         <R2ML.Terms.TerBasic.Variables.ObjectVariable
                          xmi.idref="a1"/>
                      </R2ML.Formulas.QuantifiedFormula.variables>
                      <R2ML.Formulas.QuantifiedFormula.formula>
                         <R2ML.Formulas.Implication xmi.id="a8">
                           <R2ML.Formulas.Implication.consequent>
                              <R2ML.Atoms.AtBasic.EqualityAtom xmi.id="a9"
                               isNegated="false">
                                <R2ML.Atoms.AtBasic.EqualityAtom.terms>
                                  <R2ML.Terms.ReferencePropertyFunctionTerm
                                   xmi.idref="a3"/>
                                  <R2ML.Terms.ReferencePropertyFunctionTerm
                                   xmi.idref="a5"/>
                                </R2ML.Atoms.AtBasic.EqualityAtom.terms>
                              </R2ML.Atoms.AtBasic.EqualityAtom>
                           </R2ML.Formulas.Implication.consequent>

                           <!-- ... -->

                         </R2ML.Formulas.Implication>
                       </R2ML.Formulas.QuantifiedFormula.formula>
                     </R2ML.Formulas.UniversallyQuantifiedFormula>
                  </R2ML.Rules.IntegrityRule.constraint>
              </R2ML.Rules.AlethicIntegrityRule>
         </R2ML.IntegrityRuleSet.rules>
      </R2ML.IntegrityRuleSet>
   </R2ML.RuleBase.rules>
</R2ML.RuleBase>
```

Figure 4.9: The R2ML XMI representation of the integrity rule shown in Figure 3.4 (R2ML XML) and Figure 4.5 (XML XMI)

4.2.1.2. Transforming the R2ML model into the R2ML XML Schema

Transformation of R2ML rule from R2ML abstract syntax (i.e., meta-model) into the R2ML XML concrete syntax is the transformation 2. R2ML2XML in Figure 4.2. This transformation process also consists of two primary steps as follows.

Step 1. The transformation of R2ML models to the XML models. We transform an R2ML model (RuleBase_R2ML from Figure 4.3) into an XML model (RuleBase_XML) by using an ATL transformation named R2ML2XML.atl (step 5 in Figure 4.4). After applying this transformation to the input R2ML models, XML models (RuleBase_XML) are stored in the model repository (RuleBase_XML.xmi in Figure 4.4). The output XML model conforms to the XML meta-model. The R2ML2XML.atl transformation transforms elements of the source R2ML model to elements of the target XML model. This transformation is also executed on the M1 level, as well as the XML2R2ML.atl transformation. Mappings from Table 4.1 between the R2ML XML Schema, XML meta-model, and R2ML meta-model apply here with no changes. So, for the R2ML rules given the R2ML XMI format in Figure 4.9, the result was an XML model which can be serialized back into the XML XMI format (step 6 in Figure 4.4), as it is shown in Figure 4.5.

Step 2. The XML extraction from the MOF technological space to the XML technological space. In this step, XML model (RuleBase_XML in Figure 4.4) which is generated in step 1 above and conforms to MOF-based XML meta-model, is transformed to RuleBase.xml document (Step 7 in Figure 4.4), by using the XML extractor that is a part of the ATL toolkit.

Creating a transformation from the R2ML meta-model to the R2ML XML schema (R2ML2XML), appeared to be easier to implement than XML2R2ML transformation. For the R2ML2XML transformation, only one helper is needed for the checking the negation of Atoms. All the ATL matched transformation rules are defined straightforward similar as in the XML2R2ML transformation, except for the unique elements (like `ObjectVariable`). For every R2ML meta-model element, we create a necessary number of XML meta-model elements (e.g., `Attribute`, `Element`), which corresponds to the R2ML XML Schema. Since XML meta-model `Element`'s children association end is defined as a composition (Figure 4.3), we can just not use one `ObjectVariable` and then reference to it from different `Element`-s. ATL lazy rules are used to implement support for these elements. When called, ATL lazy rules create a new element with the same content from the same input element. For example, for a unique lazy rule, which has created a unique `ObjectVariable` element (from Figure 4.8), a corresponding lazy rule is defined, like the one shown in Figure 4.10.

```
lazy rule ObjectVariable {
  from i : R2ML!ObjectVariable
  to   o : XML!Element (
             name <- 'r2ml:ObjectVariable',
             children <- Sequence { attrName,
                                    if not i.classRef.oclIsUndefined() then
                                        thisModule.ClassRule(i.classRef)
                                    else  OclUndefined
                                    endif
                         }
         ),
       attrName : XML!Attribute (
           name <- 'r2ml:name',
           value <- i.name
       )
}
```

Figure 4.10: Lazy rule of the `ObjectVariable` element, in the ATL transformation from the R2ML meta-model to the R2ML XML format

This rule creates the output XML Element with the name "`r2ml:ObjectVariable`" for every R2ML `ObjectVariable` on which it is called. In Figure 4.10, we also give an example of invocation of another lazy rule (i.e., `ClassRule`) for the R2ML `Class` elements that are also unique.

During the implementation we have created a certain number of different ATL rules and helper operation for transformations. Table 4.2 shows different number of ATL rules for every developed transformation.

Table 4.2: Number of different ATL rules in XML2R2ML and R2ML2XML transformations

Transformation/rules	**Matched rules**	**Lazy rules**	**Unique lazy rules**	**Helpers**
R2ML XML Schema to R2ML meta-model	43	0	9	28
R2ML meta-model to R2ML XML Schema	43	12	0	1

The number of matched rules is the same for both transformations. This is obvious because R2ML elements are transformed straightforward to its R2ML XML Schema representation and vice versa. As it is already described, for every element that must be defined as unique in the MOF technological space (i.e., the R2ML meta-model), unique lazy rules are used for its creation. In the opposite direction, only lazy rules have been used, since we needed to create more than one output elements with the same contents from one input element. The number of helpers is much larger in the XML2R2ML transformation, and this is due to the need to walk through the input XML model and find occurrences of the same `ObjectVariable`-s, `DataVariable`-s and `GenericVariable`-s.

4.2.2. Transformations between the SWRL XML Schema and the RDM model

Transformation between the SWRL XML Schema (for actual version 0.6) and the RDM meta-model is very similar to transformation between the R2ML XML Schema and the R2ML meta-model. These transformations also use XML meta-model and `XMLInjector` and `XMLExtractor` Java classes from ATL tools.

Figure 4.11 shows a general principle for mapping between the SWRL XML concrete syntax and the SWRL abstract syntax (i.e., meta-model). From Figure 4.11 we can see that it is possible to do transformations in both directions, as it is possible to do it with R2ML. Details about these transformations are shown in the following two sections.

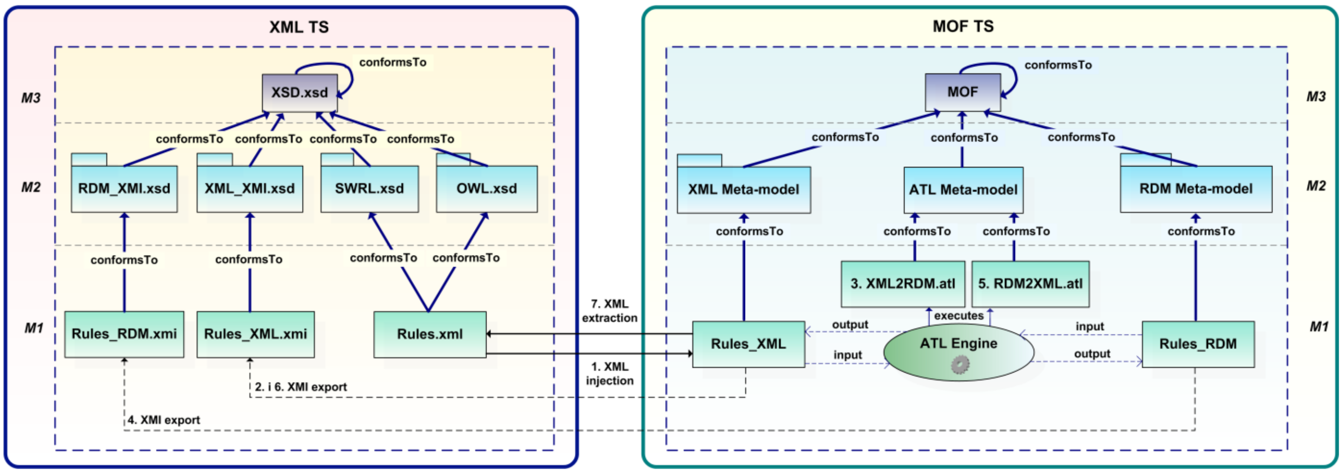


Figure 4.11: The transformation scenario: SWRL/OWL XML format into the RDM meta-model and vice

4.2.2.1. Transforming the SWRL XML Schema into the RDM model

Transformation of the SWRL rules from the SWRL XML concrete syntax into the SWRL abstract syntax (i.e., meta-model) is the transformation 3. XML2RDM in Figure 4.2. The transformation process of the SWRL (OWL) XML Schema into the R2ML meta-model consists of two primary steps as follows.

Step 1. This step consists of transformation of the SWRL/OWL rules from the XML technological space into the MOF technological space. SWRL is represented in the XML concrete syntax that represents combination of the OWL XML concrete syntax [Hori et al., 2003] with SWRL XML concrete syntax [Horrocks et al., 2004]. Transition process from the XML technological space into the MOF technological space is described in detail for the R2ML XML and R2ML meta-model in section 4.2.1. In this step, SWRL (OWL) documents (Rules.xml in Figure 4.11) are transformed in form that conforms to MOF. SWRL/OWL rules are transformed to the RDM meta-models [Brockmans & Haase, 2006], which represent the SWRL/OWL language abstract syntax in the MOF technological space.

By using XML injector from the ATL tools, SWRL XML documents are transformed to models which conform to the MOF-based XML meta-model (step 1 in Figure 4.11). A result of this process is an XML model that can be represented in the XML XMI format (step 2 in Figure 4.11). Hence, this XML model in the XML XMI format can be used as input in the ATL transformation. For SWRL rule defined in Figure 4.12, after XML injection into the MOF technological space, we attained XML model as an instance of the XML meta-model, which is represented in the XML XMI format in Figure 4.13.

```
<ruleml:Implies xmlns:owlx="http://www.w3.org/2003/05/owl-xml"
                xmlns:swrlx="http://www.w3.org/2003/11/swrlx"
                xmlns:ruleml="http://www.w3.org/2003/11/ruleml">
 <ruleml:body>
   <swrlx:individualPropertyAtom swrlx:property="hasParent">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x2</ruleml:Var>
   </swrlx:individualPropertyAtom>
   <swrlx:individualPropertyAtom swrlx:property="hasBrother">
      <ruleml:Var>x2</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
   </swrlx:individualPropertyAtom>
 </ruleml:body>
 <ruleml:head>
   <swrlx:individualPropertyAtom swrlx:property="hasUncle">
      <ruleml:Var>x1</ruleml:Var>
      <ruleml:Var>x3</ruleml:Var>
   </swrlx:individualPropertyAtom>
 </ruleml:head>
</ruleml:Implies>
```

Figure 4.12: SWRL rule in the XML-based concrete syntax

```
<!--...-->
<XML.Element xmi.id = 'a14' name = 'swrlx:individualPropertyAtom' value = ''>
      <XML.Element.children>
            <XML.Attribute xmi.id = 'a15' name = 'swrlx:property' value = 'hasParent'/>
            <XML.Element xmi.id = 'a16' name = 'ruleml:var' value = ''>
                  <XML.Element.children>
                        <XML.Text xmi.id = 'a17' name = '#text' value = 'x2'/>
                  </XML.Element.children>
            </XML.Element>
            <XML.Element xmi.id = 'a18' name = 'ruleml:var' value = ''>
                  <XML.Element.children>
                        <XML.Text xmi.id = 'a19' name = '#text' value = 'x1'/>
                  </XML.Element.children>
            </XML.Element>
      </XML.Element.children>
</XML.Element>
<!--...-->
```

Figure 4.13: The integrity rule from Figure 4.12 as an instance of the XML meta-model, represented in the XML XMI format

Step 2. In this step XML model (Rules_XML in Figure 4.11) is transformed to the RDM-based model (Rules_RDM in Figure 4.11). This transformation is executed by using the XML2RDM.atl transformation (step 3. in Figure 4.11). Output RDM model (Rules_RDM from Figure 4.11) conforms to the RDM meta-model. RDM model for the SWRL rule (see Figure 4.12) can be exported in the form of the RDM XMI documents (shown in Figure 4.14), by using NetBeans Metadata Repository (MDR) functionalities.

```
<!--...-->
<RDM.Rule xmi.id = 'a13'>
    <RDM.Rule.hasAntecedent>
          <RDM.Antecedent xmi.idref = 'a4'/>
    </RDM.Rule.hasAntecedent>
    <RDM.Rule.hasConsequent>
          <RDM.Consequent xmi.idref = 'a11'/>
    </RDM.Rule.hasConsequent>
</RDM.Rule>
<RDM.Antecedent xmi.id = 'a4'>
    <RDM.Antecedent.containsAtom>
       <RDM.Atom xmi.idref = 'a5'/>
       <RDM.Atom xmi.idref = 'a6'/>
    </RDM.Antecedent.containsAtom>
</RDM.Antecedent>
<!--...-->
<RDM.Atom xmi.id = 'a7' name = 'IndividualPropertyAtom'>
    <RDM.Atom.terms>
       <RDM.IndividualVariable xmi.idref = 'a1'/>
       <RDM.IndividualVariable xmi.idref = 'a2'/>
    </RDM.Atom.terms>
    <RDM.Atom.hasPredicateSymbol>
       <RDM.ODM.ObjectProperty xmi.idref = 'a8'/>
    </RDM.Atom.hasPredicateSymbol>
</RDM.Atom>
<RDM.ODM.ObjectProperty xmi.id = 'a8' name = 'hasUncle' deprecated = 'false'
     functional = 'false' transitive = 'false' symmetric = 'false'
     inverseFunctional = 'false' complex = 'false'/>
<RDM.IndividualVariable xmi.id = 'a1' name = 'x1'/>
<RDM.IndividualVariable xmi.id = 'a2' name = 'x3'/>
<RDM.IndividualVariable xmi.id = 'a3' name = 'x2'/>
<!--...-->
```

Figure 4.14: RDM XMI representation of the SWRL rule from Figure 4.12

In the XML2RDM.atl transformation, source elements from the XML model are transformed to the target elements of the RDM model. XML2RDM.atl transformation, same as the XML2R2ML.atl, is done on the M1 level (i.e., the model level) of the MDA, and it uses the information about elements from the M2 (meta-model) level (i.e., the XML and RDM meta-models).

Table 4.3 shows an excerpt of mappings between the SWRL XML Schema (with OWL XML Schema), XML meta-model and RDM meta-model.

Table 4.3: An excerpt of mappings between the OWL/SWRL XML schema, XML meta-model and the RDM meta-model

SWRL XML Schema	XML meta-model	RDM meta-model	Description
`individualPropertyAtom`	`Element name = 'swrlx:individualPropertyAtom'`	`Atom`	Atom which contain property name and two elements that can be individual, variable or value.
`OneOf`	`Element name = 'owlx:OneOf'`	`EnumeratedClass`	Contain individual enumeration.
`var`	`Element name = 'ruleml:var'`	`IndividualVariable`	Represents a variable.
`sameIndividualAtom`	`Element name = 'swrlx:sameIndividualAtom'`	`Atom`	Denotes equality between set of individuals or variables.
`maxcardinality`	`Element name = 'owlx:maxcardinality'`	`MaxCardinalityRestriction`	Declares restriction on maximal cardinality value.

As with XML2R2ML transformation, besides transformation of elements with matched rules, certain number of input elements is transformed by using lazy rules, e.g. for multiple *var* elements with the same attribute "name" from the SWRL to the unique `IndividualVariable` element of the RDM meta-model.

4.2.2.2. Transforming the RDM model into the SWRL XML Schema

Transformation of the SWRL rule from the SWRL abstract syntax (i.e., RDM meta-model) to the SWRL XML concrete syntax is the transformation 4. RDM2XML (see Figure 4.2). Similar as with transformation of the R2ML meta-model into the R2ML XML Schema, we created transformation from the RDM meta-model into the SWRL XML Schema (RDM2XML). This transformation process also consists of two primary steps as follows.

Step 1. The transformation of RDM models to XML models. Here we transform an RDM model (Rules_RDM from Figure 4.11) into the XML model (Rules_XML) by using an ATL transformation named RDM2XML.atl (step 5 in Figure 4.11). After applying this transformation to the input RDM models, XML models (Rules_XML) are stored in the model repository (Rules_XML.xmi in Figure 4.11). Mappings from Table 4.3 between the SWRL XML Schema, XML meta-model, and the RDM meta-model apply here with no changes. So, for the RDM rules given the RDM XMI format in Figure 4.14, we got an XML model which can be serialized back into the XML XMI format (step 6 in Figure 4.11), as it is shown in Figure 4.13.

Step 2. The XML extraction from the MOF technological space to the XML technological space. In this step, XML model (Rules_XML in Figure 4.11) which conforms to the MOF-based XML meta-model, and is generated in step 1 above, is transformed to the Rules.xml document (Step 7 in Figure 4.11).

This transformation, as well as R2ML2XML, includes one helper operation which returns all restriction defined on the certain property. RDM meta-model elements are transformed to the elements of XML meta-model by using ATL matched rules. The exception are the unique transformation elements (as it is `IndividualVariable`), for which we use lazy rules.

During implementation of these transformations, a certain number of different ATL rules and helper operations have been created. Table 4.4 shows a different number of ATL rules for every transformation.

Table 4.4: Number of different ATL rules in the XML2RDM and RDM2XML transformations

Transformation/rules	**Matched rules**	**Lazy rules**	**Unique lazy rules**	**Helpers**
SWRL XML Schema to RDM meta-model	35	0	3	23
RDM meta-model to SWRL XML Schema	35	3	0	1

As with transformation of the R2ML meta-model into the R2ML XML Schema, number of matched rules is the same for both transformations. As it is already described for the R2ML, for every element which must be defined as unique in the MOF technological space (i.e., RDM meta-model), we used unique lazy rules for their creation. In the opposite direction we used only lazy rules, because we had to create more than one output element with the same contents from the same input element. The number of helper operations is larger in the XML2RDM transformation, because we need to search (recursively) through the input XML model to find instances of the same `IndividualVariable` elements.

4.3. TRANSFORMATIONS BETWEEN THE R2ML MODEL AND THE RDM MODEL

In the previous section we presented how we had bridged the XML and the MOF technological spaces, i.e., how it was possible to get RDM or R2ML models from the SWRL/OWL and R2ML rules defined in the XML technological space, respectively. Transformations between the R2ML model and the RDM model are based on conceptual mappings which are defined for these transformations of elements that belong to the concrete syntax of these languages, i.e., R2ML (version 0.5) and SWRL/OWL (version 0.6), which is shown in Table 4.5. In this table, acronym CD represents class description from the OWL XML concrete syntax [Hori et al., 2003], i.e. expression which is used as basic constitutive block of class definitions. This class description in OWL XML concrete syntax can be represented by one of the following elements: `Class`, `DataRestriction`, `ObjectRestriction`, `UnionOf`, `IntersectionOf`, `ComplementOf` and `OneOf`. Expression ***T***(expr1, expr2, ...) is the translation operator of SWRL element into the R2ML element, while *t* is a variable.

Table 4.5: Conceptual mappings between the SWRL and R2ML

***SWRL* expression**	***R2ML* expression**
SWRL Class atoms	
ClassAtom(classID, t)	ObjectClassificationAtom(classID, t)
ClassAtom(UnionOf(CD1, CD2), t)	Disjunction(***T***(ClassAtom(CD1, t)), ***T***(ClassAtom(CD2, t)))
ClassAtom(IntersectionOf(CD1, CD2), t)	Conjuction(***T***(ClassAtom(CD1, t)), ***T***(ClassAtom(CD2, t)))
ClassAtom(ComplementOf(CD), t)	StrongNegation(***T***(ClassAtom(CD, t)))
ClassAtom(OneOf({objID1,...,objIDn}), t)	Disjunction(EqualityAtom(objID1, t),..., EqualityAtom(objIDn ,t))
ClassAtom(ObjectRestriction(objPropID, allValuesFrom(CD)), t)	UniversallyQuantifiedFormula(x, Implication(ReferencePropertyAtom(objPropID, t, x), ***T***(ClassAtom(CD, t)))
ClassAtom(ObjectRestriction(objPropID, someValuesFrom(CD)), t)	ExistentiallyQuentifiedFormula (x, Conjuction(***T***(ClassAtom(CD, t)), ReferencePropertyAtom(objPropID, t, x)))
ClassAtom(ObjectRestriction(objPropID, hasValue(objID)), t)	ReferencePropertyAtom(objPropID, t, objID)
ClassAtom(ObjectRestriction(objPropID, mincardinality(n)), t)	AtLeastQuantifiedFormula(n, x, ReferencePropertyAtom(objPropID, t, x))
ClassAtom(ObjectRestriction(objPropID, maxcardinality(n)), t)	AtMostQuantifiedFormula(n, x, ReferencePropertyAtom(objPropID, t, x))
ClassAtom(ObjectRestriction(objPropID, mincardinality(m), maxcardinality(n)), t)	AtLeastAndAtMostQuantifiedFormula(m, n, x, ReferencePropertyAtom(objPropID, t, x))
ClassAtom(ObjectRestriction(objPropID, cardinality(n)), t)	AtLeastAndAtMostQuantifiedFormula(n, n, x, ReferencePropertyAtom(objPropID, t, x))
SWRL Datarange atoms	
DatarangeAtom(datatypeID, t)	DataClassificationAtom(datatypeID, t)
DatarangeAtom(OneOf({objID1,...,objIDn}), t)	Disjunction(DatatypePredicateAtom(swrlb:equal, objID1, t), ... DatatypePredicateAtom(swrlb:equal, objIDn, t))
DatarangeAtom(DataRestriction(dataPropID, allValuesFrom(dataTypeID)), t)	UniversallyQuantifiedFormula(x, Implication(AttributionAtom(dataPropID, t, x), ***T***(DatarangeAtom(datatypeID, t)))
DatarangeAtom(DataRestriction(dataPropID, someValuesFrom(datatypeID)), t)	ExistentiallyQuentifiedFormula(x, Conjuction(***T***(DatarangeAtom(datatypeID, t)), AttributionAtom(dataPropID, t, x))
DatarangeAtom(DataRestriction(dataPropID, allValuesFrom(OneOf({dataLiteral}))), t)	UniversallyQuantifiedFormula(x, Implication(AttributionAtom(dataPropID, t, x), ***T***(DatarangeAtom(OneOf({objID1, ..., objIDn}), t))
DatarangeAtom(DataRestriction(dataPropID, someValuesFrom(OneOf({dataLiteral}))), t)	ExistentiallyQuentifiedFormula(x, Conjuction(***T***(DatarangeAtom(OneOf({objID1, ..., objIDn}), t)), AttributionAtom(dataPropID, t, x))
DatarangeAtom(DataRestriction(dataPropID, hasValue(dataLiteral)), t)	AttributionAtom(dataPropID, t, dataLiteral)
DatarangeAtom(DataRestriction(dataPropID, mincardinality(n)), t)	AtLeastQuantifiedFormula(n, x, AttributionAtom(dataPropID, t, x))
DatarangeAtom(DataRestriction(dataPropID, maxcardinality(n)), t)	AtMostQuantifiedFormula(n, x, AttributionAtom(dataPropID, t, x))
DatarangeAtom(DataRestriction(dataPropID, mincardinality(m), maxcardinality(n)), t)	AtLeastAndAtMostQuantifiedFormula(m, n, x, AttributionAtom(dataPropID, t, x))
SWRL object property atoms	
IndividualvaluedPropertyAtom(objectID1, objectID2)	ReferencePropertyAtom(individualvaluedPropertyID, objectID1, objectID2)
SWRL datatype property atoms	
DatavaluedPropertyAtom(objectID, dataLiteral)	AttributionAtom(datavaluedPropertyID, objectID, dataLiteral)
SWRL equality and inequality atoms	
SameAs(objectID1, objectID2)	EqualityAtom(objectID1, objectID2)
DifferentFrom(objectID1, objectID2)	InequalityAtom(objectID1, objectID2)
SWRL Built-in atoms	
BuiltIn(builtinID, t)	DatatypePredicateAtom (builtinID, t)

Figure 4.15 shows a general transformation scenario between the RDM model and the R2ML model by using ATL. From this figure we can see that it is possible to execute transformations in both directions. Details about these transformations are shown in the following two sections.

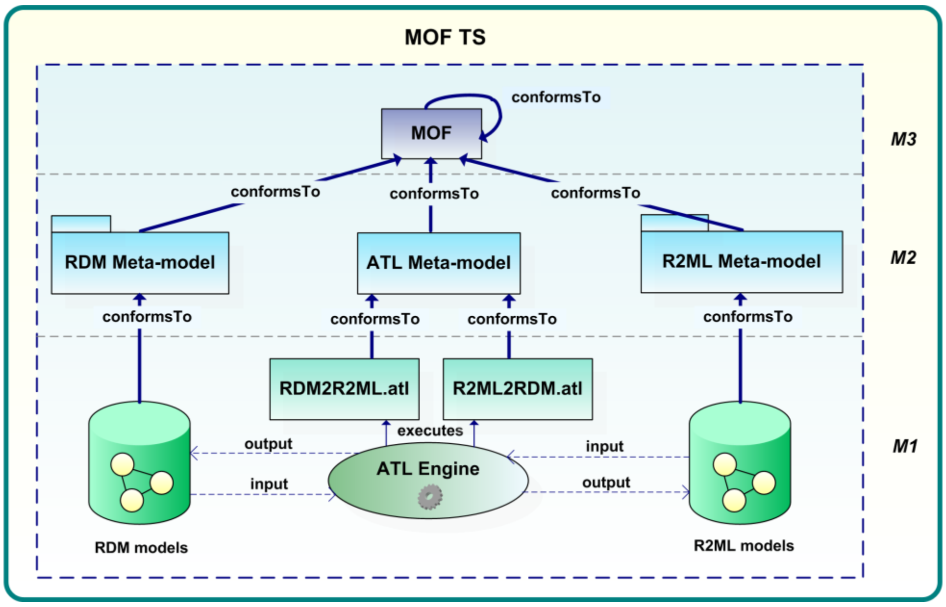


Figure 4.15: Transformation of the RDM model elements into the R2ML model elements and vice versa

4.3.1. Transforming the RDM model into the R2ML model

Transformation of a model in the SWRL abstract syntax (i.e., RDM meta-model) into a model in the R2ML abstract syntax (i.e., R2ML meta-model) is the transformation 5. RDM2R2ML in Figure 4.2. Transformation implementation between the RDM model and the R2ML model is based on the conceptual mappings of the concrete syntaxes of these models, SWRL/OWL and R2ML XML. The transformation process between RDM model elements into the R2ML model elements is done in the MOF technological space. Table 4.6 shows an excerpt of mappings between the SWRL XML Schema, XML meta-model, RDM meta-model and R2ML meta-model.

Table 4.6: Mappings between the SWRL/OWL XML Schema, XML meta-model, RDM meta-model and R2ML meta-model

SWRL/OWL	XML meta-model	RDM meta-model	R2ML meta-model
individualPropertyAtom	Element name = 'swrlx:individualProperty Atom'	Atom	UniversallyQuantifiedFormula
OneOf	Element name = 'owlx:OneOf'	EnumeratedClass	Disjunction
var	Element name = 'ruleml:var'	IndividualVariable	ObjectVariable
sameIndividualAtom	Element name = 'swrlx:sameIndividualAtom'	Atom	EqualityAtom
maxcardinality	Element name = 'owlx:maxcardinality'	MaxCardinality Restriction	AtMostQuantifiedFormula

For the SWRL/OWL XML Schema complex type, an instance of the XML meta-model is created, through a transformation from the XML technological space into the MOF technological space, as it is described in section 4.2.2.1. Hence, these XML elements are transformed to the instances of the RDM meta-model by using ATL, and then to the instances of the R2ML meta-model.

The transformation between RDM meta-model elements and R2ML meta-model elements is defined as a sequence of the ATL rules and helper operations (RDM2R2ML.atl in Figure 4.15). In this ATL transformation we used matched and unique lazy rules. ATL transforms the input RDM meta-model elements into the output R2ML meta-model elements. Figure 4.16 shows an example of one matched rule from the RDM2R2ML.atl transformation. This example is made for the `IndividualPropertyAtom` class of the RDM meta-model.

```
rule IndividualPropertyAtom2ReferencePropertyAtom{
      from i : RDM!Atom (
                        i.name = 'IndividualPropertyAtom'
               )
      to refpropat : R2ML!ReferencePropertyAtom (
                        isNegated <- false,
                        referenceProperty <- i.hasPredicateSymbol,
                        subject <- thisModule.IndividualVariable2ObjectVariable( i.terms->last() ),
                        object <- thisModule.IndividualVariable2ObjectVariable( i.terms->first() )
                     )
}
```

Figure 4.16: A matched rule that transforms an RDM `IndividualPropertyAtom` to an R2ML `ReferencePropertyAtom`

In the RDM2R2ML.atl transformation, besides ATL rules, we have defined helper operations that are used from rules. An example of such operation is used for transforming the RDM `Class` element to the R2ML `ObjectClassificationAtom` element. In the rules which transform input RDM `Class` element, there is no direct reference to the RDM `ClassAtom`, so it is not possible to know to which rule (or multiple input rules) above mentioned `Class` element belongs. The basic advantage of the ATL is that this operation instead of `Class` element can be called for any `Class` element subclasses. To solve this situation we defined an ATL helper operation that returns a reference to the required `ClassAtom`, shown in Figure 4.17.

```
helper def: getAtomForClassElement(cla : RDM!Class) : RDM!Atom =
     let allAtoms : Sequence(RDM!Atom) = RDM!Atom.allInstancesFrom('IN')->asSequence()
                                              ->select(c | c.hasPredicateSymbol.oclIsKindOf(RDM!Class)) in
          allAtoms->iterate(p; res : RDM!Atom = allAtoms->first() |
               if(p.childrenClasses(cla)->includes(cla))
                    then p
                    else res
                    endif
          );
```

Figure 4.17: ATL helper operation that founds RDM `Atom` for the required RDM `Class` element

In this transformation we have applied a special type of the ATL rules, *endpoint* rule that is executed once all other rule have been executed on the elements of input model. This rule does not match any element of the input model. Figure 4.18 shows an *endpoint* rule that, after transforming all RDM `Rule` elements to the R2ML `AlethicIntegrityRule` elements, comprise those elements to the `IntegrityRuleSet`, apropos the `RuleBase`.

```
endpoint rule RuleBase() {
      to rb : R2ML!RuleBase (
                     rules <- Sequence { rs }
               ),
         rs : R2ML!IntegrityRuleSet (
                     rules <- RDM!Rule.allInstancesFrom('IN')->asSequence()
               )
}
```

Figure 4.18: ATL *endpoint* rule for the R2ML `RuleBase` element

After execution of the RDM2R2ML transformation on the RDM model from Figure 4.14, R2ML model is attained, and it is shown in Figure 4.19 in the R2ML XMI format.

```
<!-- ... -->
<R2ML.Formulas.UniversallyQuantifiedFormula xmi.id = 'a4'>
   <R2ML.Formulas.QuantifiedFormula.variables>
       <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a7'/>
       <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a5'/>
       <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a6'/>
   </R2ML.Formulas.QuantifiedFormula.variables>
   <R2ML.Formulas.QuantifiedFormula.formula>
      <R2ML.Formulas.Implication xmi.id = 'a8'>
         <R2ML.Formulas.Implication.consequent>
           <R2ML.Atoms.AtRelational.ReferencePropertyAtom xmi.id = 'a9' isNegated = 'false'>
             <R2ML.Atoms.AtRelational.ReferencePropertyAtom.object>
               <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a6'/>
             </R2ML.Atoms.AtRelational.ReferencePropertyAtom.object>
             <R2ML.Atoms.AtRelational.ReferencePropertyAtom.referenceProperty>
               <R2ML.Vocabulary.VocBasic.ReferenceProperty xmi.idref = 'a10'/>
             </R2ML.Atoms.AtRelational.ReferencePropertyAtom.referenceProperty>
             <R2ML.Atoms.AtRelational.ReferencePropertyAtom.subject>
               <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a5'/>
             </R2ML.Atoms.AtRelational.ReferencePropertyAtom.subject>
           </R2ML.Atoms.AtRelational.ReferencePropertyAtom>
         </R2ML.Formulas.Implication.consequent>
         <!-- ... -->
      </R2ML.Formulas.Implication>
   </R2ML.Formulas.QuantifiedFormula.formula>
</R2ML.Formulas.UniversallyQuantifiedFormula>
<!-- ... -->
```

Figure 4.19: R2ML XMI representation of the RDM rule shown in Figure 4.14

The R2ML model (an instance of the R2ML meta-model) which is obtained in the previously described process, can be serialized to the R2ML XML concrete syntax by using the R2ML2XML transformation and the ATL's XML extractor (described in section 4.2.1.2), and it is shown in Figure 4.20.

```
<!--...-->
<r2ml:UniversallyQuantifiedFormula>
      <r2ml:ObjectVariable r2ml:name = 'x1'/>
      <r2ml:ObjectVariable r2ml:name = 'x2'/>
      <r2ml:ObjectVariable r2ml:name = 'x3'/>
      <r2ml:Implication>
          <r2ml:consequent>
             <r2ml:ReferencePropertyAtom r2ml:referencePropertyID = 'hasUncle'>
               <r2ml:subject>
                 <r2ml:ObjectVariable r2ml:name = 'x3'/>
               </r2ml:subject>
               <r2ml:object>
                 <r2ml:ObjectVariable r2ml:name = 'x1'/>
               </r2ml:object>
             </r2ml:ReferencePropertyAtom>
          </r2ml:consequent>
          <--...-->
       </r2ml:Implication>
</r2ml:UniversallyQuantifiedFormula>
<!--...-->
```

Figure 4.20: R2ML XML representation (an excerpt) of rule shown in Figure 4.19

4.3.2. Transforming the R2ML model into the RDM model

The transformation of a model in the R2ML abstract syntax (i.e., R2ML meta-model) into a model in the SWRL abstract syntax (i.e., RDM meta-model) is transformation 6. R2ML2RDM in Figure 4.2. The transformation of R2ML model elements to the RDM model elements is based on the same mappings from Table 4.5, between abstract syntaxes of these meta-models. The mappings from Table 4.6 between the R2ML meta-model elements and RDM meta-model elements, apply here with no changes. It should be noticed that in Table 4.5 mappings between SWRL concrete syntax and R2ML abstract syntax (i.e., meta-model) are shown, while in Table 4.6 mappings of elements from SWRL concrete syntax, via XML meta-model and SWRL abstract syntax (i.e., RDM meta-model) to R2ML meta-model are shown. The transformation process of the R2ML model to the RDM model is done in MOF technological space, by using the R2ML2RDM.atl transformation (see Figure 4.15). After applying this transformation to the input R2ML models, RDM models are stored in the model repository. For the R2ML XML rules shown in Figure 4.20, by using the XML injector and XML2R2ML transformation we attained R2ML XMI model in R2ML XMI format (see Figure 4.19), and by using the R2ML2RDM transformation that R2ML XMI model can be transformed to the RDM model shown in the RDM XMI format (see Figure 4.14).

In the R2ML2RDM transformation we defined rules for transforming certain R2ML elements (e.g., `ExistentiallyQuantifiedFormula`) into the RDM model elements, in case when this element is transformed to the RDM Atom (e.g., `ClassAtom`), when it belongs to Implication element which is constrained by `IntegrityRule`. We also defined two helper operations, first (`isConjuctionFromIntersection`) which is shown in Figure 4.12, that checks for certain Conjuction element if that element is from `IntersectionOf` SWRL construction, or it is just set of atoms.

```
helper context R2ML!Conjuction def: isConjuctionFromIntersection : Boolean =
     if R2ML!ExistentiallyQuantifiedFormula.allInstancesFrom('IN')->asSequence()->collect(c | c.formula)
          ->flatten()->excludes(self)
          and  self.formulas->size() = 2
          and self.formulas->asSequence()->first().getTerm = self.formulas->asSequence()->last().getTerm
          and ( self.formulas->select(c | c.oclIsTypeOf(R2ML!Disjunction))->collect(e | e.formulas)->flatten()
               ->forAll(c | c.oclIsTypeOf(R2ML!EqualityAtom))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!UniversallyQuantifiedFormula))->collect(e | e.formula)
               ->flatten()->forAll(c | c.oclIsTypeOf(R2ML!Implication))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!ExistentiallyQuantifiedFormula))->collect(e | e.formula)
               ->flatten()->forAll(c | c.oclIsTypeOf(R2ML!Conjuction))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!AtLeastQuantifiedFormula))->collect(e | e.formula)->flatten()
               ->forAll(c | c.oclIsTypeOf(R2ML!ReferencePropertyAtom))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!AtMostQuantifiedFormula))->collect(e | e.formula)->flatten()
               ->forAll(c | c.oclIsTypeOf(R2ML!ReferencePropertyAtom))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!AtLeastAndAtMostQuantifiedFormula))->collect(e | e.formula)
               ->flatten()->forAll(c | c.oclIsTypeOf(R2ML!ReferencePropertyAtom))
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!AttributionAtom))->size() = 0
          and self.formulas->select(c | c.oclIsTypeOf(R2ML!DatatypePredicateAtom))->size() = 0 )
          and self.formulas->collect(c | c.oclIsTypeOf(R2ML!ReferencePropertyAtom))->flatten()->size() > 0
     then true
     else false
     endif;
```

Figure 4.21: ATL helper operation `isConjuctionFromInteresction`

The second helper operation is called `isRefPropAtomForHasValue`, and it implements the checking algorithm for the R2ML `ReferencePropertyAtom` element (returns Boolean, i.e., logical value true or false), i.e. it checks whether that R2ML `ReferencePropertyAtom` is candidate for transformation to the RDM `HasValueRestriction` element (see Figure 4.22).

```
helper context R2ML!ReferencePropertyAtom def:
            isRefPropAtomForHasValue : Boolean =
      thisModule.getChildrenOfAllTopMostImplications()->excludes(self) and
      R2ML!UniversallyQuantifiedFormula.allInstancesFrom('IN')
                        ->excluding(thisModule.getTopMostUniversallyQuantifiedFormulas())
                                      ->select(e | e.formula.oclIsTypeOf(R2ML!Implication))
                                                    ->collect(c | c.formula)->flatten()
                                                              ->collect(c | c.antecedent)
                                                                       ->excludes(self) and
      R2ML!ExistentiallyQuantifiedFormula.allInstancesFrom('IN')
                         ->select(e | e.formula.oclIsTypeOf(R2ML!Conjuction))
                                                   ->collect(c | c.formula.formulas->asSequence())
                                                                       ->excludes(self) and
      R2ML!AtMostQuantifiedFormula.allInstancesFrom('IN')
                                     ->collect(c | c.formula)->flatten()->excludes(self) and
      R2ML!AtLeastQuantifiedFormula.allInstancesFrom('IN')
                                     ->collect(c | c.formula)->flatten()->excludes(self) and
      R2ML!AtLeastAndAtMostQuantifiedFormula.allInstancesFrom('IN')
                                        ->collect(c | c.formula)->flatten()->excludes(self);
```

Figure 4.22: ATL helper operation `isRefPropAtomForHasValue`

Table 4.7 shows the number of ATL rules and helper operations for every transformation which is discussed in this section. The number of matched and unique lazy rules in transformation of the R2ML model to the RDM model is larger then in transformation of the RDM meta-model to the R2ML meta-model, because of much bigger expressivity of R2ML. This means that it is possible to transform certain R2ML elements to the RDM elements, which are not attained by direct transformation from the RDM model, but from some other model. The number of helper operations is the same, because in case of both transformations it is necessary to search through input model elements.

Table 4.7: Number of different ATL rules in the RDM2R2ML and R2ML2RDM transformations

Transformation/rules	**Matched rules**	**Lazy rules**	**Unique lazy rules**	**Helpers**
RDM meta-model to R2ML meta-model	42	1	11	31
R2ML meta-model to RDM meta-model	53	0	15	31

4.4. TRANSFORMATIONS BETWEEN THE R2ML MODEL AND OCL MODEL

Transformations between the R2ML model elements and OCL model elements are executed completely in the MOF technological space. These transformations are based on conceptual mappings which are defined for transformations of these languages abstract syntax elements, the R2ML and OCL, respectively, which is shown in Table 4.8 (for OCL we used concrete syntax for simpler representation). We have used the following acronyms in the table:

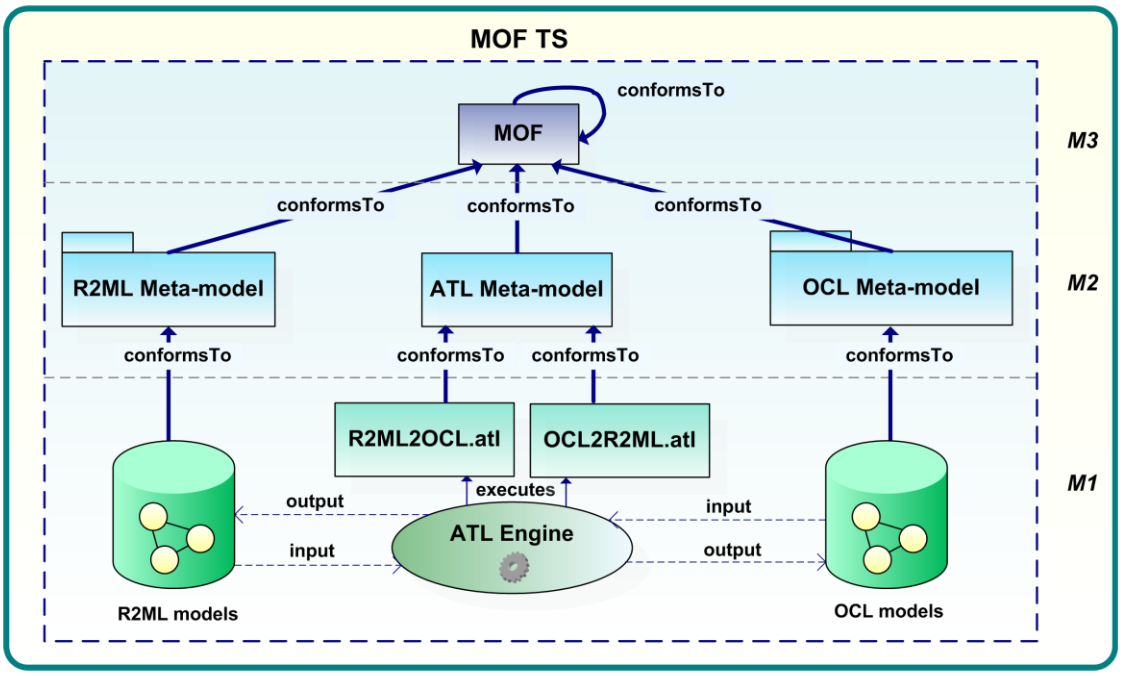
- rOp - represents OCL relational operator (<, >, >=, <=, <>, =);
- expr - OCL expression (`OCLExpression` class from OCL meta-model);
- pType - primitive OCL datatype (can be: `String`, `Integer`, `Real`, `Boolean`);
- aEnd - UML association end;
- attr - attribute;
- iVar - iterator variable;
- collectionFunction - predefined OCL function on collection, which can be: *select*, *reject*, *collect*, *forAll*, *exists* or *iterate*.

The source of the OCL expression on top is always contextual `Class` element or "self" variable of type `Variable`. Expression ***T***(expr1, expr2, ...) represents the translation operator of OCL element to the R2ML element.

Table 4.8: Conceptual mappings between the OCL and R2ML

OCL expression	***R2ML expression***
[not] expr (rOP) pType	DatatypePredicateAtom([not] ***T***(rOP), ***T***(expr), ***T***(pType))
[self.] attr (rOp)	AttributeFunctionTerm(***T***(attr.source)) or AttributionAtom(***T***(attr.source)) in case when rOp = "=" and pType
Context Class invariant	UniversallyQuantifiedFormula(ObjectVariable(Class), ***T***(invariant))
pType	TypedLiteral(***T***(pType))
expr.attr	AttributeFunctionTerm(ReferencePropertyFunctionTerm (***T***(expr.source)))
expr1 (or) expr2	Disjunction(***T***(expr1), ***T***(expr2))
expr1 (and) expr2	Conjuction(***T***(expr1), ***T***(expr2))
attr = pType	AttributionAtom(attr, attr.source, ***T***(pType))
expr1.expr2 = expr3	ReferencePropertyAtom(ReferencePropertyFunctionTerm(expr2), ReferencePropertyFunctionTerm(expr1), ReferencePropertyFunctionTerm(expr3))
expr1.expr2	ReferencePropertyFunctionTerm(expr2, ReferencePropertyFunctionTerm(expr1)) if "parent" is Atom or AttributeFunctionTerm(expr2, ReferencePropertyFunctionTerm(expr1)), if "parent" is Term
expr1 (implies) expr2	Implication(***T***(expr1), ***T***(expr2))
expr.function()	DataOperationTerm(function, ***T***(expr))
[not] expr->collectionFunction()	DatatypePredicateAtom([neg], collectionFunction, ***T***(expr))
expr->collectionFunction() = pType	DatatypePredicateAtom(collectionFunction, ***T***(pType))
expr->collectionFunction().attr = pType	DatatypePredicateAtom(equal, AttributeFunctionTerm (ObjectOperationTerm(ReferencePropertyFunctionTerm(expr.source))))
expr = expr1->collectionFunction()	DatatypePredicateAtom(***T***(expr), ***T***(expr1->collectionFunction()))
expr2->collectionFunction() (rOP) pType	DatatypePredicateAtom(DataTypeFunctionTerm(collectionFunction, ***T***(expr2)), ***T***(pType))
expr.isKindOf(Class) ili expr.isTypeOf(Class)	ObjectClasssificationAtom(***T***(expr))
expr1->select (expr2) [(rOP) ili collectionFunction (rOP) pType]	Conjuction(***T***(expr1), ExistentiallyQuantifiedFormula(Conjuction(GenericAtom(iVar), DatatypePredicateAtom(expr2))))
expr1->includesAll(expr2)	Implication(GenericAtom(GenericVariable, ***T***(expr1)), GenericAtom(GenericVariable, ***T***(expr2)))
expr1->forAll(expr2)	Conjuction(***T***(expr1), ***T***(expr2))
iVar1 (rOp) iVar2	EqualityAtom([neg], ***T***(iVar1), ***T***(iVar2))

Figure 4.23 shows the transformation scenario between the R2ML model and the OCL model, by using ATL. From this figure we can see that it is possible to do transformations in both directions. Details about these transformations are shown in the following two sections.



Slika 4.23: The transformation scenario: the R2ML model into the OCL model and vice versa

4.4.1. Transforming the R2ML model into the OCL model

Transformation of a model in the R2ML abstract syntax (i.e., R2ML meta-model) into a model in the OCL abstract syntax (i.e., OCL meta-model) is transformation 7. R2ML2OCL in Figure 4.2. The implementation of this transformation is based on conceptual mappings of the concrete syntaxes of these meta-models, the R2ML and OCL, respectively.

The transformation of the R2ML models to the OCL models is done by using the R2ML2OCL.atl transformation, as it is shown in Figure 4.23. Input R2ML models conforms to the R2ML meta-model, while output OCL models conform to the OCL meta-model. Table 4.9 show an excerpt of mappings between the R2ML meta-model elements and the OCL meta-model elements, which is based on conceptual mappings between these languages abstract syntaxes, which is shown in Figure 4.8.

Table 4.8: An excerpt of mappings between the R2ML meta-model elements and the OCL meta-model elements

R2ML meta-model	**OCL meta-model**
RuleBase	OclModule
AlethicIntegrityRule	Invariant
Conjuction	OperatorCallExp (name = 'and')
Implication	OperatorCallExp (name = 'implies')
AttributionAtom	OperatorCallExp (name = '=') source = PropertyCallExp (subject)
ObjectVariable	VariableExp
EqualityAtom	OperatorCallExp (name = '=') source = Iterator
ReferencePropertyFunctionTerm	PropertyCallExp referredProperty (name = 'property') source = VariableExp
ObjectOperationTerm	CollectionOperationCallExp

In the R2ML2OCL.atl transformation we have implemented the transformation of the R2ML integrity rules into the OCL invariants. The context of such invariants in the OCL meta-model is represented with the `OclContextDefinition` class, where contextual element `Class` is from the R2ML `UniversallyQuantifiedFormula`, which is formula in the R2ML `AlethicIntegrityRule` element, which is transformed by the ATL rule `AlethicIntegrityRule2Invariant` (see Figure 4.24). The content of the OCL `Invariant` is a formula transformed from the R2ML `AlethicIntegrityRule` element, except in the case when that formula is R2ML `Implication` which contains R2ML `ReferencePropertyAtom`'s. In that case the R2ML `UniversallyQuantifiedFormula` is transformed to the OCL `IteratorExp` element (*forAll*).

```
rule AlethicIntegrityRule2Invariant {
   from i : R2ML!AlethicIntegrityRule (
                 i.oclIsTypeOf(R2ML!AlethicIntegrityRule)
            )
  to o : OCL!Invariant (
              name <- if not i.ruleID.oclIsUndefined() then
                               i.ruleID
                      else OclUndefined
                       endif,
              contextDefinition <- contextDef,
              specification <- if i.constraint.formula.oclIsTypeOf(R2ML!Implication)
                               then
                                  -- In case of implication for forAll
                                  if i.constraint.formula.getChildren->forAll(c |
                                                    c.oclIsTypeOf(R2ML!ReferencePropertyAtom)) then
                                             thisModule.UnivQuantFormImpl2ForAllIteratorExp(i.constraint)
                                  else i.constraint.formula
                                  endif
                               else i.constraint.formula
                               endif
          ),
      contextDef : OCL!OclContextDefinition (
              contextElement <- if not i.constraint.variables->asSequence()
                                                ->first().classRef.oclIsUndefined() then
                                      thisModule.Class2Class(i.constraint.variables->asSequence()
                                                                                      ->first().classRef)
                                  else thisModule.ObjectVariable2Class(i.constraint.variables
                                                                                 ->asSequence()->first())
                                  endif
              )
}
```

Figure 4.24: ATL rule which transforms the R2ML `AlethicIntegrityRule` into the OCL `Invariant` element

When the R2ML2OCL.atl transformation (shown in Figure 4.23) is executed on input R2ML model (in R2ML XMI format shown in Figure 4.29), we get OCL model represented in the OCL XMI format, as shown in Figure 4.25.

```
<OCL.OclModule xmi.id = 'a1'>
     <OCL.OclModule.ownedElements>
       <OCL.Invariant xmi.id = 'a2'>
         <OCL.OclModuleElement.contextDefinition>
           <OCL.OclContextDefinition xmi.id = 'a3'>
             <OCL.OclContextDefinition.contextElement>
               <OCL.Class xmi.id = 'a4' name = 'Person' isAbstract = 'false'/>
             </OCL.OclContextDefinition.contextElement>
           </OCL.OclContextDefinition>
         </OCL.OclModuleElement.contextDefinition>
         <OCL.Invariant.specification>
           <OCL.IteratorExp xmi.id = 'a5' name = 'forAll'>
             <OCL.CallExp.source>
               <OCL.OperationCallExp xmi.id = 'a6' name = 'allInstances'>
                 <OCL.CallExp.source>
                   <OCL.VariableExp xmi.id = 'a7'>
                     <OCL.VariableExp.referredVariable>
                       <OCL.Variable xmi.idref = 'a8'/>
                     </OCL.VariableExp.referredVariable>
                   </OCL.VariableExp>
                 </OCL.CallExp.source>
               </OCL.OperationCallExp>
             </OCL.CallExp.source>

            <!--...-->

            </OCL.IteratorExp>
         </OCL.Invariant.specification>
       </OCL.Invariant>
     </OCL.OclModule.ownedElements>
</OCL.OclModule>
```

Figure 4.25: OCL XMI representation of transformed R2ML rule in the R2ML XMI format from Figure 4.19

4.4.2. Transforming the OCL model into the R2ML model

Transformation of a model in the OCL abstract syntax (i.e., OCL meta-model) into a mdel in the R2ML abstract syntax (i.e., R2ML meta-model) is transformation 8. OCL2R2ML in Figure 4.2. The transformation of the OCL model elements into the R2ML model elements is based on the same conceptual mappings of these languages from Table 4.8, and they apply here with no changes. The transformation process is executed in the MOF technological space, by using the OCL2R2ML.atl transformation (see Figure 4.23). After applying this transformation to the input OCL models, the resulting R2ML models are stored in the model repository. For the input OCL model shown in the OCL XMI format in Figure 4.25, we got the output R2ML shown in the R2ML XMI format in figure 4.19.

The OCL2R2ML transformation transforms OCL invariants (`Invariant` elements) into the R2ML integrity rules (`AlethicIntegrityRule` elements), which is shown in the ATL rule in Figure 4.26. In the created R2ML `AlethicIntegrityRule` element, the constraint is the R2ML `UniversallyQuantifiedFormula`. In this formula, the R2ML `ObjectVariable` is attained by transforming contextual class (`Class` element) from the OCL `Invariant` element. If the defined constraint in the OCL `Invariant` is `IteratorExp`, then formula in the `UniversallyQuantifiedFormula` is located directly to that element. A special case is the `OperatorCallExp` (*not*) OCL element, which is bypassed, and we go directly to source of that element.

```
rule Invariant2AlethicIntegrityRule {
      from i : OCL!Invariant(
                  i.oclIsTypeOf(OCL!Invariant)
            )
      to o : R2ML!AlethicIntegrityRule (
                  constraint <- uqf,
                  ruleID <- if not i.name.oclIsUndefined() then
                                    i.name
                            else OclUndefined
                            endif
            ),
         uqf : R2ML!UniversallyQuantifiedFormula (
                  variables <- if i.specification.oclIsTypeOf(OCL!IteratorExp) then
                                          Sequence { thisModule.Class2ObjectVariable(i.contextDefinition.contextElement),
                                                     i.specification.iterator->collect(c | thisModule.Iterator2ObjectVariable(c))
                                                            ->flatten()
                                                   }
                              else thisModule.Class2ObjectVariable(i.contextDefinition.contextElement)
                              endif,
                  formula <- if i.specification.oclIsTypeOf(OCL!OperatorCallExp) and i.specification.name = 'not' then
                                    i.specification.source  -- jump over [not]
                             else if i.specification.oclIsTypeOf(OCL!IteratorExp) and i.specification.name = 'forAll' then
                                         if i.specification.source.oclIsTypeOf(OCL!OperationCallExp) then
                                                if i.specification.source.name = 'allInstances' then
                                                      i.specification.body
                                                else OclUndefined
                                                endif
                                         else i.specification
                                         endif
                                  else if i.specification.oclIsKindOf(OCL!CallExp) then -- check if we have IteratorExp, and
                                                if i.specification.haveIteratorExpInSources() then -- if is go to it
                                                      i.specification.getIteratorExpFromSources()
                                                else i.specification
                                                endif
                                       else i.specification
                                       endif
                                  endif
                             endif
            )
}
```

Figure 4.26: ATL rule which transforms the OCL `Invariant` into the R2ML `AlethicIntegrityRule`

Table 4.9 shows the number of different ATL rules and helper operations for every transformation. The number of matched and lazy rules in transformation of the R2ML model into the OCL model is some larger, similar as in transformation of the R2ML model into the RDM model, because of the larger R2ML expressivity. The number of helper operations is larger in the transformation of the OCL into the R2ML model, because of more complex lookup of the OCL meta-model elements.

Table 4.9: The number of different ATL rules in the OCL2R2ML and R2ML2OCL transformations

Transformation/rules	**Matched rules**	**Lazy rules**	**Unique lazy rules**	**Helpers**
OCL meta-model to R2ML meta-model	26	0	11	11
R2ML meta-model to OCL meta-model	28	2	8	8

4.5 TRANSFORMATIONS BETWEEN TEXTUAL AND MOF-BASED OCL REPRESENTATION

Because OCL concrete syntax is defined in the EBNF technological space [OCL, 2006], it is necessary to enable transformation of OCL rules in representation that conforms to the OCL meta-model from the MOF technological space. In this thesis, bridging EBNF and MOF technological spaces for OCL language is done by using Textual Concrete Syntax (TCS) software environment as part of the ATL tools [Jouault et al., 2006]. TCS represents *domain specific language* (DSL) for defining textual concrete syntaxes in Model Driven Engineering (MDE). TCS can be used for parsing text-to-model and serialization model-to-text.

The overview of the usage of the TCS language is shown in Figure 4.27. First, we expressed OCL meta-model in the KM3 format[31] (MOF technological space). The definition of the OCL concrete syntax is expressed in TCS and is denoted as OCL_TCS model. The bridge between the EBNF and MOF technological spaces consists of the EBNF injector and extractor (more precisely, the `EBNFInjector` and `EBNFExtractor` Java classes as part of ATL tools). The injector takes an OCL invariant (model) expressed in the textual concrete syntax of the OCL and generates a model conforming to the OCL meta-model in the MOF technological space. An example of OCL invariant in the OCL textual concrete syntax is denoted as "OCL code" in Figure 4.27, in the EBNF technological space on M1 level, and it conforming to the OCL grammar, which is denoted as OCL.g on M2 level. This grammar is expressed in the ANTLR (ANother Tool for Language Recognition)[32] [Parr, 2005]. The extractor creates a textual representation of the model from the MOF technological space that conforms to the OCL meta-model. Figure 4.27 shows extraction from the OCL model into the OCL code.

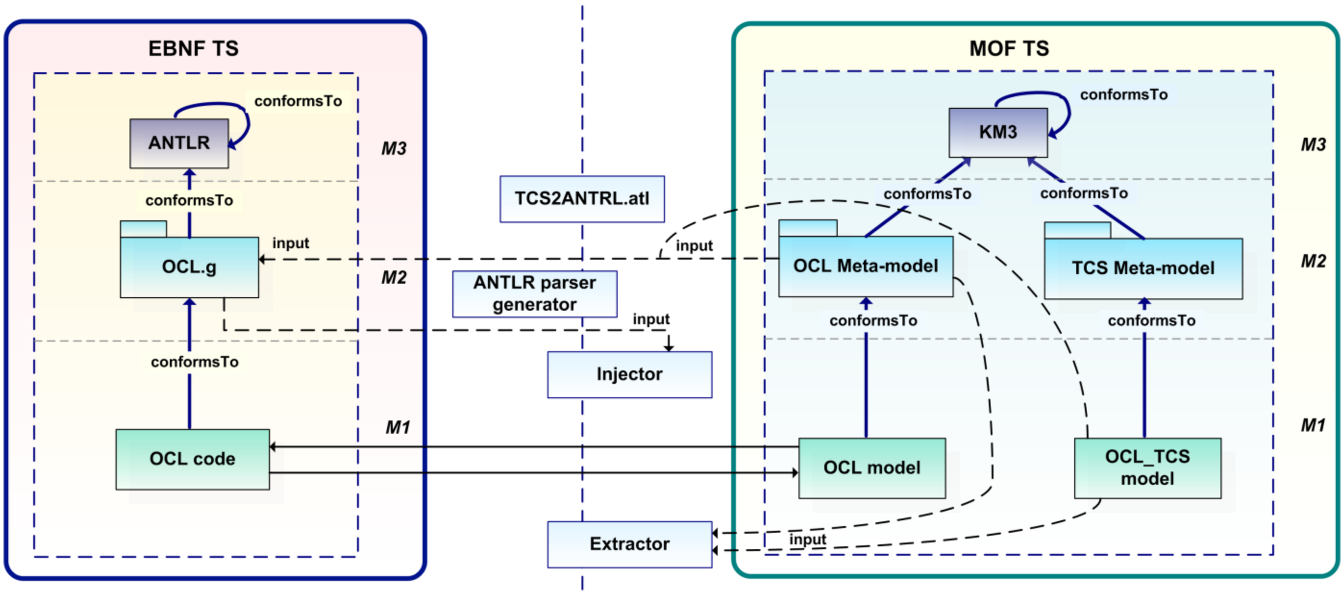


Figure 4.27: Using TCS for bridging the EBNF and MOF technological spaces for OCL

The approach that TCS uses is the one that starts from the OCL meta-model and its concrete textual syntax description. The goal is to obtain three entities for OCL: its annotated grammar OCL.g expressed in ANTLR, and the couple of injector and extractor. This annotated grammar contains not only grammar rules, but also code (i.e., semantic action), which will be executed by generated parser when that parser comes to a certain point in the annotated grammar. Annotated grammars can be also regarded as operators, denominates which symbols ANTLR tool will regard as sub-branch roots of the generated parsed OCL code abstract tree, which will be regarded as leaves, and that will be ignored in conformity with construction of that tree. ANTLR takes the annotated grammar in a textual form and creates a parser for it. The OCL.g grammar is created by using the TCS2ANTLR.atl transformation (implements a set of declarative translation rules), which is developed as part of the TCS tools [Jouault et al., 2006]. This transformation takes the OCL meta-model and OCL_TCS model (OCL concrete syntax described with TCS meta-model) as input (shown with dashed lines in Figure 4.27) and generates the rules and the annotations in the OCL grammar. This grammar is used to generate the

[31] KM3 is a domain specific language (DSL) for defining meta-model as the MOF or ECore, and it is very similar to Java syntax [Jouault & Bézivin, 2006].
[32] ANother Tool for Language Recognition (ANTLR) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. ANTLR provides support for tree construction, tree walking, and translation. [Online] http://www.antlr.org/.

EBNF injector. The injector is a parser (`OCLParser` class) generated by the tools provided by the ANTLR technology. Besides the `OCLParser` class, ANTLR creates the `OCLLexer` class, also, which precedes the parser and it is used for reading all input OCL code characters and building vocabulary symbols for parser. This generation is done by the ANTLR parser generator, shown in Figure 4.27.

On the other side, the EBNF extractor, as part of TCS tools, works on the internal model representation expressed in the OCL and creates its textual representation. The EBNF extractor takes an OCL model, the OCL meta-model and description of the OCL textual concrete syntax in TCS (OCL_TCS model in Figure 4.27), and it creates OCL concrete textual representation from the OCL MOF-based models.

4.5.1. Transforming the OCL model into the OCL concrete syntax

The transformation of the OCL abstract syntax (i.e., OCL meta-model) into the OCL concrete syntax is transformation 9. OCL meta-model to OCL Text, in Figure 4.2. The mapping between the OCL meta-model and OCL concrete syntax is based on the defined OCL textual concrete syntax in the TCS language. Figure 4.28 shows an excerpt of the mappings between the OCL meta-model (in the KM3 format, Figure 4.28a) and its textual concrete syntax defined in TCS (Figure 4.28b).

```
class OclModule extends Package {
        reference ownedElements[*] ordered container : OclModuleElement;
    }


class Invariant extends OclModuleElement {
        attribute name : String;
        reference specification container : OclExpression;
    }


class OclContextDefinition extends Element {
        reference contextElement container : Class;
    }
```

a) OCL meta-model

```
template OclModule main
        : ownedElements
        ;

template Invariant context
        :  (isDefined(contextDefinition) ? contextDefinition)
           <newline> <tab> "inv" (isDefined(name) ? name)
           <no_space> ":" [ specification ] { endNL = false}
        ;

template OclContextDefinition
        : "context" contextElement
        ;
```

b) OCL textual concrete syntax

Figure 4.28: An excerpt of transforming the OCL meta-model elements into the OCL textual concrete syntax (TCS)

Figure 4.28 shows the corresponding part of the OCL meta-model, because TCS works in a way that it defines for every meta-model element its description in the textual concrete syntax, by using the constructs of this language (e.g. *template*). Figure 4.28 shows the following elements:

- Class `OclModule` is represented with `ownedElement`'s (of type `OclModuleElement`) and is a construct which is first created (denoted with *main*[33]).
- Class `Invariant` is represented with: context definition (i.e. name of the appropriate UML or MOF class), keyword "inv", the name of this constraint if it is defined, the symbol ":' and the specification (body) of this constraint. Elements <newline> and <tab> are used in serialization of OCL model to OCL code, and they represent a new line feed, and text bias, respectively.
- Class `OclContextDefinition` is represented with the keyword "context" and `contextElement` element.

TCS elements are associated by name to their corresponding elements in the meta-model. For example, TCS template `OclModule` corresponds to the KM3 class `OclModule`, while `ownedElement` corresponds to the KM3 `OclModuleElement` class (and all its subclasses). These examples show that it is straightforward to encode a simple syntax in TCS: syntactic elements are specified in syntax order.

[33] An important constraint imposed by the TCS on meta-models is that they must have a root element.

Because of TCS-specific constraints and requirements (and ANTLR before all), we changed the OCL meta-model according to such requirements. We defined package `EnhancedOCL` (see section 3.3.4.3), where we added the `OperatorCallExp` element for expressions that use these two following operators: the `CollectionOperationCallExp` for invoking operations on collections and `Iterator` class which is special type of iterator variable. We also added two classes for every OCL primitive type, e.g. `StringType` or `IntegerType`. We introduced the `DefOclModuleElement` for "def" elements (for operations `OclOperation` class, and for attributes `OclProperty` class), `Invariant` class for "inv" elements, and the `DeriveOclModuleElement` for "derive" elements. These three types inherit abstract class `OclModuleElement`, which contains context definition (`OclContextDefinition` class), while the `OclContextDefinition` contains `Class` element as contextual element. The `OclModule` element is class that is added in order to capture different `OclModuleElement`'s.

For the OCL model in the OCL XMI format shown in Figure 4.25, by using the TCS meta-model, defined OCL textual concrete syntax and the EBNF extractor (as it is described in previous section), we get the OCL code that is shown in Figure 4.29.

```
context Person
   inv: Person.allInstances()->forAll(x1, x2 | (self.hasFather = x1 and
                                                x1.hasBrother = x2) implies
                                                self.hasUncle = x2)
```

Figure 4.29: OCL rule in the textual concrete syntax that represents transformed OCL model shown in Figure 4.25

4.5.2. Transforming the OCL concrete syntax into the OCL model

The transformation of the OCL concrete syntax into the OCL abstract syntax (i.e., OCL meta-model) is transformation 10. OCL Text to OCL meta-model in Figure 4.2. Transforming OCL code to OCL model (also called *parsing*) is done by using the generated ANTLR `OCLLexer` and `OCLParser` classes, as well as the EBNF injector. This injector takes for input the OCL meta-model, an OCL code that we want to parse (as *.ocl* textual file), the generated `OCLLexer` and `OCLParser` classes, and for output it returns an OCL model in the OCL XMI format. For the OCL rule that is shown in Figure 4.29, we get an OCL model which can be represented in the OCL XMI format, as it is shown in Figure 4.25.

In parsing OCL code into the OCL model, we used the same textual concrete syntax as shown in Figure 4.28. However, here we have one constraint, i.e. it is possible to represent only MOF composition relation between elements. By using only composition, models are constrained in a way, that only one element can refer to the same class, and this is a problem because we must have multiple unique elements in a model (e.g., variables). However, this can be achieved by using associations (references) which enable that one unique element can be refered by multiple elements. TCS has a construct which is called the symbol table that enables this. Using the symbol table for representing references is shown in an example of a variable in the OCL below.

In the OCL meta-model, `VariableExp` class is defined as `OclExpression` which contains reference that is not composition (with name `referredVariable`) to the `Variable` class. This relation is shown in Figure 4.30a (in the KM3 format). To enable the creation of unique `Variable` element on the basis of variable name, this element in the OCL textual concrete syntax is denoted as `addToContext` (see Figure 4.30b). This means that every time when a `Variable` is recognized in the code, it is added to the symbol table. The reference `referredVariable` in the `VariableExp` element that should refer to `Variable` elements is denoted with `refersTo = name`. This means that every time when a `Variable` is recognized in code, a `VariableExp` will be created and connected with this `Variable` via the name attribute. That `Variable` will be searched in the symbol table. Each class template in TCS can specify the creation of a new symbol table. This is declared using the `context` keyword in the declaration of a template. In this way we secured that `Variable` that is defined in some `Invariant` would not be used in another. `Invariant` element is denoted as contextual element (see Figure 4.28b).

```
class VariableExp extends OclExpression {
          reference referredVariable[0-1] : Variable;
}

class Variable extends TypedElement {
          reference initExpression[0-1] container : OclExpression;
}
```

a) Variables in OCL meta-model

```
template VariableExp
          : referredVariable{refersTo = name}
          ;

template Variable addToContext
          : name (isDefined(type) ? ":" type)
            (isDefined(initExpression) ? "=" initExpression)
          ;
```

b) Variables in TCS

Figure 4.30: Variable definition in the OCL meta-model and its corresponding textual concrete syntax

The previous procedure applies when an OCL code is parsed to an OCL model, and it is shown on the example of `Variable` and `VariableExp`, i.e. as the OCL model that is shown in OCL XMI format in Figure 4.25, and in concrete syntax in Figure 4.29. For clearer view, OCL model from Figure 4.25 is shown as UML object model in Figure 4.31. In this figure we can see that references (associations) are shown with dashed lines. We also see that `Variable` and `Iterator` elements that are refered by the `VariableExp` element using the `referredVariable` association, are unique in the whole model.

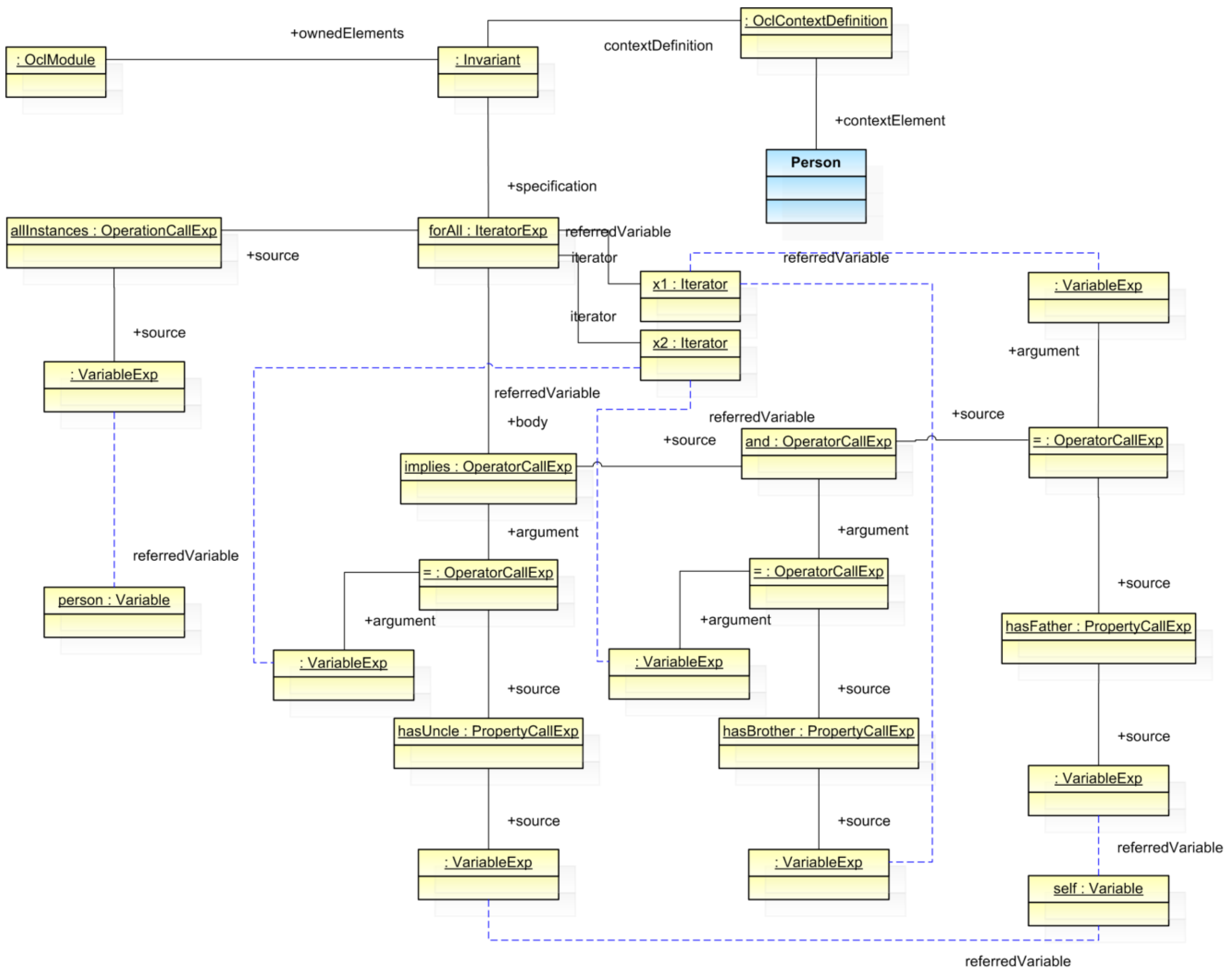


Figure 4.31: UML object model of the OCL rule shown in the OCL XMI format in Figure 4.25

5. IMPLEMENTATION, TESTING AND USING R2ML TRANSFORMATIONS

This chapter presents implementation details of the model-to-model (M2M) transformation between SWRL and OCL using R2ML as the central (general) rule markup language. In the first part of the chapter we have shown the ATLAS Developments tools architecture, while in the second part we have shown the implementation of transformations between SWRL rules and OCL rules. In addition, we have presented how we have integrated the ATL execution of rule transformations in a Java Web application.

5.1. IMPLEMENTATION DETAILS

Transformations between rule language models (M2M) are implemented in ADT (ATL Development Tools) [ATL, 2007] [AM3, 2007], which is a part of the Eclipse environment developed in the scope of the ATL Eclipse/GMT subproject [GMT, 2006]. The ADT tools are realized in Java programming language, as a part of an open source community, whereby platform independence, system modularity and possibility for system updates and adoptions is accomplished. Development in these tools is based on using the ATL transformation language [ATL, 2006].

Rule language meta-models (i.e., R2ML, RDM and OCL) are drawn in the form of class diagrams using the following tools: Poseidon for UML, uml2mof[34] as well as MagicDraw and Microsoft Visio tools. All of these tools (except Visio) have a possibility for importing models in UML XMI format. In addition, Magic Draw UML 11.6 has a possibility to export models in UML2 EMF XMI 2.0 format, which subsequently can be converted into ECore format by using UML2 plug-in for Eclipse.

5.1.1. ATLAS Development Tools (ADT)

The ATL language is supported by a set of tools developed on top of the Eclipse platform. Figure 5.1 shows ADT architecture.

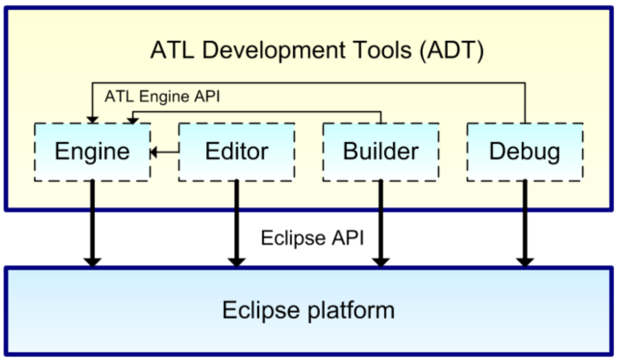
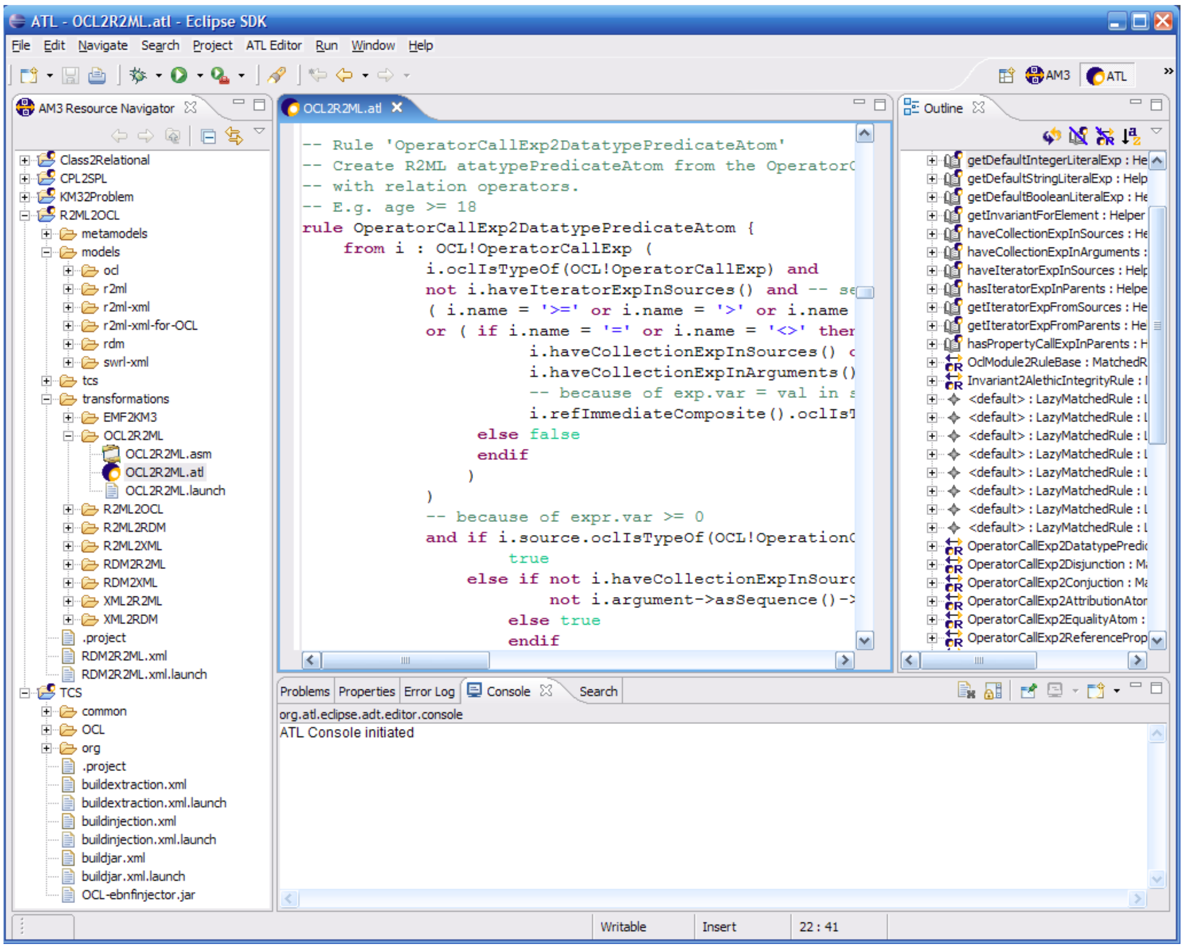


Figure 5.1: ATL Development Tools architecture [Allilaire et al., 2006]

ADT is composed of the ATL transformation engine (*Engine* block) and the ATL Integrated Development Environment (IDE: *Editor*, *Builder* and *Debug* blocks), which is shown in Figure 5.2. Complete usage documentation is provided in the ATL user manual [ATL, 2006]. The left window (AM3 Resource Navigator) displays projects in the workspace, directories and files (models, meta-models, transformations, etc.). The main window (the central one) shows the currently open file with appropriate syntax highlighting, which depends of the file type. The right window (Outline) shows

[34] Transforms input UML XMI format into the MOF XMI format, i.e., UML diagram from M1 level of the MDA architecture, translate to the M2 level.

information about the currently opened file, for example, in case an ATL file is opened, rules and helpers for that file (model) in abstract syntax, are shown. In bottom part is Console window, which shows different ATL console messages (e.g., errors in ATL code when compiling *.atl* source file to the *.asm* file which is executable in the ATL engine).


ATL - OCL2R2ML.atl - Eclipse SDK
File Edit Navigate Search Project ATL Editor Run Window Help
AM3 ATL
AM3 Resource Navigator
Class2Relational
CPL2SPL
KM32Problem
R2ML2OCL
metamodels
models
ocl
r2ml
r2ml-xml
r2ml-xml-for-OCL
rdm
swrl-xml
tcs
transformations
EMF2KM3
OCL2R2ML
OCL2R2ML.asm
OCL2R2ML.atl
OCL2R2ML.launch
R2ML2OCL
R2ML2RDM
R2ML2XML
RDM2R2ML
RDM2XML
XML2R2ML
XML2RDM
.project
RDM2R2ML.xml
RDM2R2ML.xml.launch
TCS
common
OCL
org
.project
buildextraction.xml
buildextraction.xml.launch
buildinjection.xml
buildinjection.xml.launch
buildjar.xml
buildjar.xml.launch
OCL-ebnfinjector.jar
OCL2R2ML.atl
Outline
getDefaultIntegerLiteralExp : He
getDefaultStringLiteralExp : Help
getDefaultBooleanLiteralExp : He
getInvariantForElement : Helper
haveCollectionExpInSources : He
haveCollectionExpInArguments :
haveIteratorExpInSources : Help
hasIteratorExpInParents : Helpe
getIteratorExpFromSources : He
getIteratorExpFromParents : He
hasPropertyCallExpInParents : H
OclModule2RuleBase : MatchedR
Invariant2AlethicIntegrityRule : I
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
<default> : LazyMatchedRule : L
OperatorCallExp2DatatypePredic
OperatorCallExp2Disjunction : M
OperatorCallExp2Conjuction : Ma
OperatorCallExp2AttributionAtor
OperatorCallExp2EqualityAtom :
OperatorCallExp2ReferenceProp
Problems Properties Error Log Console Search
Writable Insert 22 : 41

```
-- Rule 'OperatorCallExp2DatatypePredicateAtom'
-- Create R2ML atatypePredicateAtom from the Operator(
-- with relation operators.
-- E.g. age >= 18
rule OperatorCallExp2DatatypePredicateAtom {
    from i : OCL!OperatorCallExp (
            i.oclIsTypeOf(OCL!OperatorCallExp) and
            not i.haveIteratorExpInSources() and -- se
            ( i.name = '>=' or i.name = '>' or i.name
            or ( if i.name = '=' or i.name = '<>' ther
                    i.haveCollectionExpInSources() c
                    i.haveCollectionExpInArguments()
                    -- because of exp.var = val in s
                    i.refImmediateComposite().oclIsT
                else false
                endif
               )
            )
            -- because of expr.var >= 0
            and if i.source.oclIsTypeOf(OCL!Operation(
                    true
                else if not i.haveCollectionExpInSourc
                        not i.argument->asSequence()->
                    else true
                    endif
```

```
org.atl.eclipse.adt.editor.console
ATL Console initiated
```

Figure 5.2: ATL Eclipse-based integrated development environment

The ATL engine is responsible for dealing with core ATL tasks: compilation and execution of a transformation [ATLVM, 2005] [ATL, 2007]. ATL transformations (*.atl* source files) are compiled into programs in a specialized byte-code (*.asm* files). The byte-code is executed by the ATL Virtual Machine (VM) which is specialized for handling models and provides a set of instructions for model manipulation. The architecture of ATL execution engine is shown in Figure 5.3.

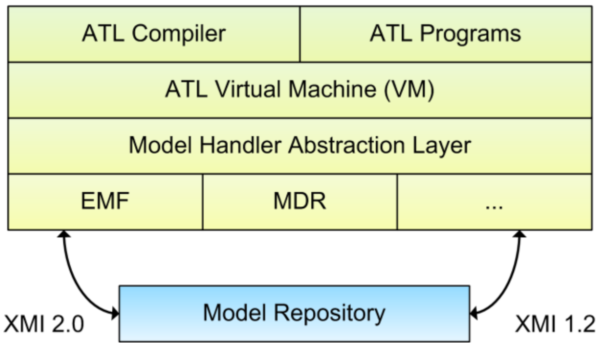
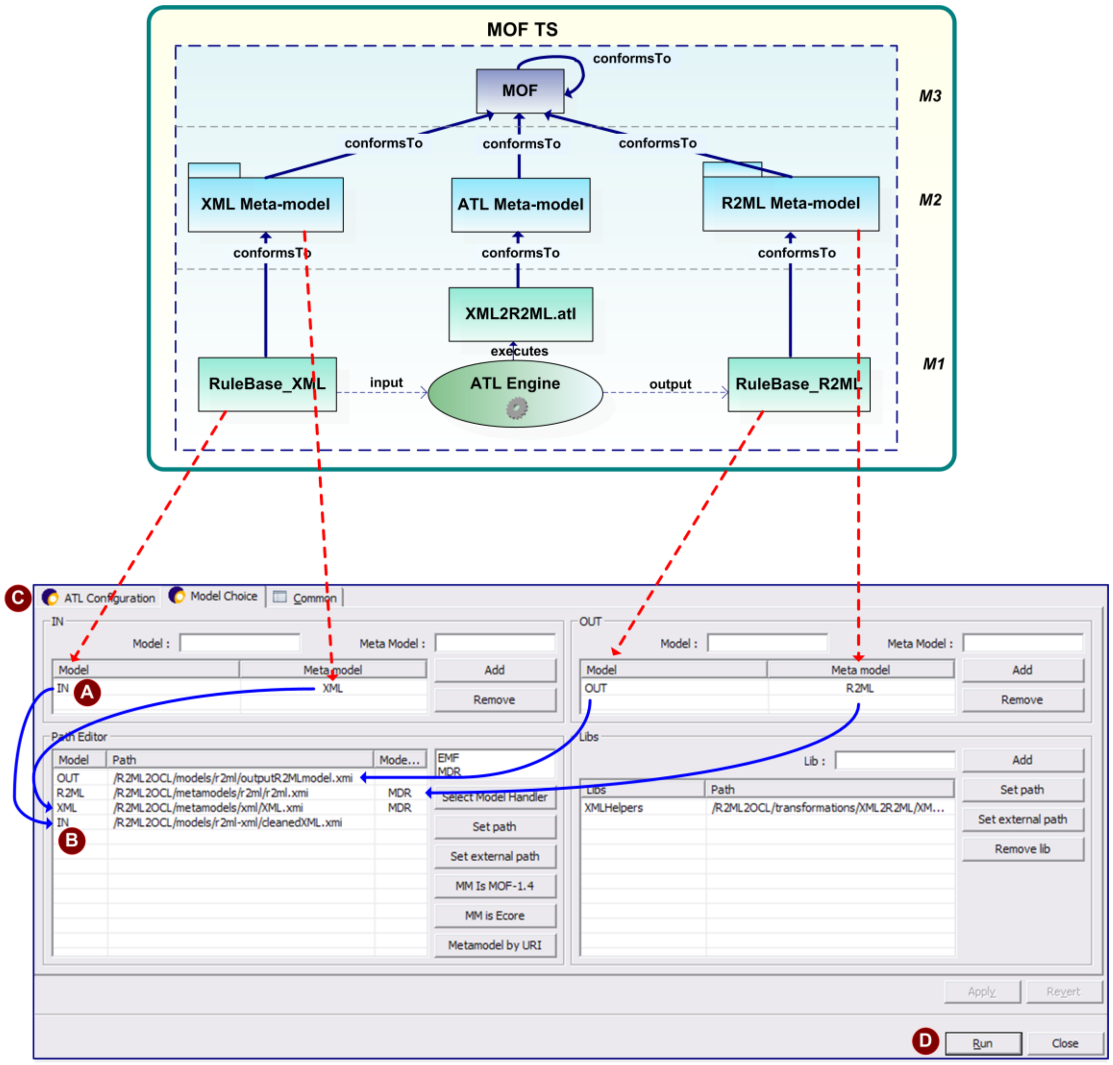


Figure 5.3: The architecture of the ATL execution engine [Allilaire et al., 2006]

The ATL Virtual Machine (VM) runs on top of various model management systems (i.e., model handlers). Model handlers are components that provide programming interface for model manipulation – Eclipse Modeling Framework[35], NetBeans Metadata Repository (MDR)[36] and AIR MDS[37] [Djurić, 2006], to name but a few. A model repository provides storage facilities for models. To make the VM independent of model handlers, an intermediary level, called Model Handler Abstraction Layer is introduced. This layer translates the model manipulation instructions of the VM to the instructions of a specific model handler.

The ATL compiler is automatically called on each ATL source file in each ATL project during the Eclipse build process. By default, this process is triggered when an ATL source file is modified (e.g., saved). The execution of an ATL transformation requires that the declared source and target models as well as meta-models are bound to the actual models (i.e., XMI files typically ending in *.xmi* or *.ecore*). This is supposed to be set in the launch configuration wizard, shown in Figure 5.4



IN

Model :
Meta Model :

Model	Meta model
IN	XML

Path Editor

Model	Path	Mode...
OUT	/R2ML2OCL/models/r2ml/outputR2MLmodel.xmi	
R2ML	/R2ML2OCL/metamodels/r2ml/r2ml.xmi	MDR
XML	/R2ML2OCL/metamodels/xml/XML.xmi	MDR
IN	/R2ML2OCL/models/r2ml-xml/cleanedXML.xmi	

EMF
MDR

OUT

Model :
Meta Model :

Model	Meta model
OUT	R2ML

Libs

Lib :

Libs	Path
XMLHelpers	/R2ML2OCL/transformations/XML2R2ML/XM...

Figure 5.4: An ATL transformation launch configuration (based on [Allilaire et al., 2006])

Figure 5.4 gives an overview of the ATL launch configuration wizard and shows the correspondence between the wizard's user interface and the operational context of an ATL transformation. The dashed arrows map models and meta-models to their declarations, whereas the

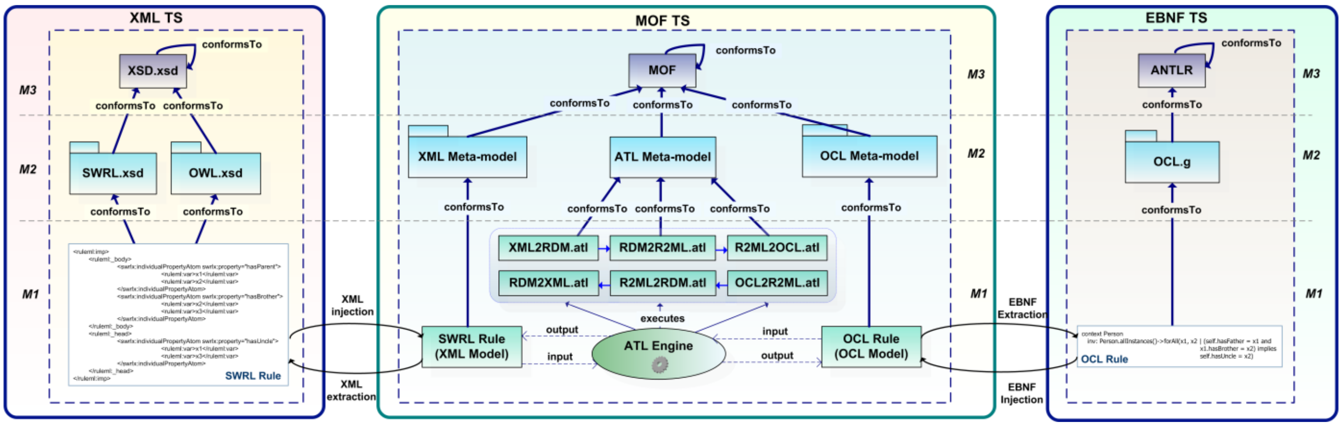
[35] http://www.eclipse.org/emf/.
[36] http://mdr.netbeans.org/.
[37] https://air-mds.dev.java.net/.

solid arrows map the declarations to the corresponding files. In XML2R2ML transformation configuration (transformation 1. XML2R2ML from Figure 4.2), which transforms an XML model into the corresponding R2ML model, the source model RuleBase_XML is declared as the model IN with XML as its meta-model (A in Figure 5.4). The file that corresponds to the IN model is cleanedXML.xmi (B in Figure 5.4). The tab “ATL Configuration” (denoted with C in Figure 5.4) is used to specify the location of the transformation to be executed. The execution can be started by clicking on the "Run" button in the bottom of the window (D in Figure 5.4), or by using Eclipse "Launch" button.

The ATL engine delegates reading and writing models to the underlying model handler. When, for instance, EMF is used, source and target models must be in EMF XMI 2.0 format, while MDR models must be in MOF XMI 1.2 format (for MOF-1.4). EMF models can be edited with the EMF reflective editor, which represents models as trees. More complex transformation scenarios can use other kinds of formats (e.g., XML, textual).

5.2. THE IMPLEMENTATION OF TRANSFORMATION RULES

The complete transformation scenario between the SWRL and OCL languages, via the R2ML language, on implementation level is shown in Figure 5.5



```
<ruleml:imp>
    <ruleml:_body>
        <swrlx:individualPropertyAtom swrlx:property="hasParent">
            <ruleml:var>x1</ruleml:var>
            <ruleml:var>x2</ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:individualPropertyAtom swrlx:property="hasBrother">
            <ruleml:var>x2</ruleml:var>
            <ruleml:var>x3</ruleml:var>
        </swrlx:individualPropertyAtom>
    </ruleml:_body>
    <ruleml:_head>
        <swrlx:individualPropertyAtom swrlx:property="hasUncle">
            <ruleml:var>x1</ruleml:var>
            <ruleml:var>x3</ruleml:var>
        </swrlx:individualPropertyAtom>
    </ruleml:_head>
</ruleml:imp>
```

```
context Person
  inv: Person.allInstances()->forAll(x1, x2 | (self.hasFather = x1 and
                                  x1.hasBrother = x2) implies
                                  self.hasUncle = x2)
```

Figure 5.5: The transformation scenario between SWRL and OCL

A SWRL rule transforms from the XML technological space into the MOF technological space by using XML injector, as an XML model. The attained XML model transforms to an RDM model with XML2RDM.atl transformation, and that RDM model is then transformed to the corresponding R2ML model by using RDM2R2ML.atl transformation. This R2ML model can be serialized in the R2ML XML format (as described in 4.2.1.2). The R2ML model is subsequently transformed into an OCL model by using R2ML2OCL.atl transformation. Finally, the OCL model can be transformed from the MOF technological space to the corresponding OCL code in the EBNF technological space by using EBNF extractor and defined TCS for OCL (textual concrete syntax).

In the opposite direction, an OCL rule in concrete syntax (from EBNF technological space) is transformed into an OCL model (the MOF technological space) by using EBNF injector and generated OCL parser. Such OCL model can be transformed to the R2ML model by using OCL2R2ML.atl transformation, which in turn can be transformed to an RDM model (with R2ML2RDM.atl transformation shown in Figure 5.5). The RDM model gets transformed into an XML model, which is then serialized to the SWRL concrete syntax (XML technological space) by using XML extractor.

5.2.1. Implementing a transformation for integrity rules

The transformation between the OCL invariants and the SWRL rules via the R2ML integrity rules is implemented in two directions, from OCL to SWRL, and from SWRL to OCL.

5.2.1.1. OCL to SWRL

The transformation of an OCL rule into the R2ML rule, and subsequently into the SWRL rule, starts from the UML class in the class diagram (see Figure 5.6), on which the OCL invariant is defined.

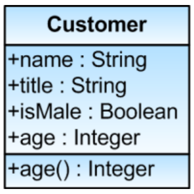


Figure 5.6: UML Customer class

An OCL invariant is defined on the class Customer, as shown in Figure 5.7

```
context Customer
   inv: self.name = 'Jos senior' implies self.age() > 21
```

Figure 5.7: An OCL invariant on the class Customer

The OCL invariant shown in Figure 5.7 is defined on the `Customer` class context shown in Figure 5.6. It represents an implication which obligates all `Customer` class instances having the "*Jos senior*" value for the name attribute, to also have a value greater than 21 as the result of invoking their "*age*" operation. It is possible to transform only OCL invariants which have implication in their body to SWRL rules, because SWRL rules are represented in the form of implications [Horrocks et al., 2004].

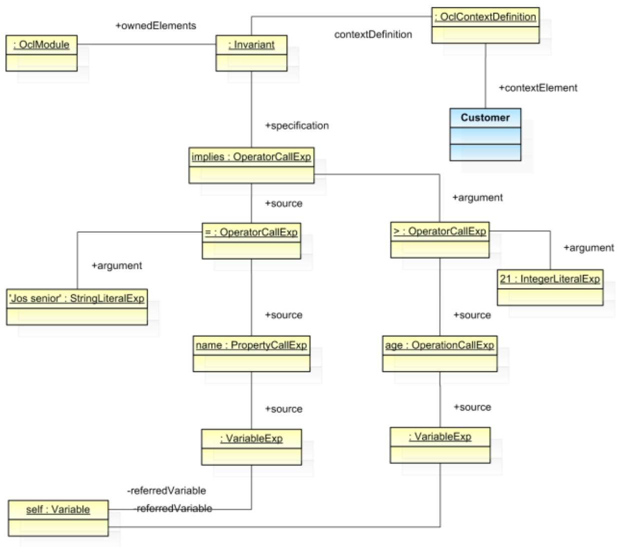
The parsing of OCL code into an OCL model (Figure 5.7) can be done in two ways. The first way is to use automatized AM3 Ant task[38] (similar to batch files), which enables model loading, transforming, parsing, as well as its serialization into XMI format. The second way is to do the parsing directly from Java code, by using ATL `EBNFInjector` class and the generated ANTLR-based classes `OCLLexer` and `OCLParser`, as it is described in section 4.5.2, and in Figure 4.2 (10. OCL Text to OCL meta-model). AM3 Ant task (buildinjection.xml) which creates OCL model from OCL code is shown in Figure 5.8.

[38] http://wiki.eclipse.org/index.php/AM3_Ant_Tasks

```
<project name="project" default="injection">

      <!-- Properties definition -->
      <property name="metamodel.name" value="OCL"/> <!-- Meta-model -->
      <property name="model.name" value="m1"/>     <!-- Model -->
      <property name="model.suffix" value=".ocl"/>  <!-- Extension -->

      <property name="project.name" value="/TCS/"/>
      <property name="metamodel.dir" value="${metamodel.name}/"/>
      <property name="metamodel.path" value="${project.name}${metamodel.dir}"/>
      <property name="ebnfinjector.name" value="ebnfinjector"/>
      <property name="ebnfinjector.dir" value="${ebnfinjector.name}/"/>
      <property name="metamodel.jar" value="${metamodel.name}-${ebnfinjector.name}.jar"/>
      <property name="metamodel.uri" value="${metamodel.path}${metamodel.name}.xmi" />
      <property name="model.uri" value="${metamodel.path}${model.name}${model.suffix}" />
      <property name="ebnfnjector.uri" value="${project.name}${metamodel.jar}" />
      <property name="classname.prefix" value="${metamodel.name}" />
      <property name="metamodel.path" value="${project.name}${metamodel.dir}"/>

      <!-- Targets definition -->
      <target name="injection">
            <am3.loadModel modelHandler="MDR" name="${metamodel.name}" metamodel="MOF" path="${metamodel.uri}" />

            <am3.loadModel modelHandler="MDR" name="${model.name}" metamodel="${metamodel.name}"
                           path="${project.name}${metamodel.dir}${model.name}${model.suffix}">
                  <injector name="ebnf">
                        <param name="name" value="${classname.prefix}"/>
                        <classpath>
                              <pathelement location="${ebnfnjector.uri}"/>
                        </classpath>
                  </injector>
            </am3.loadModel>

            <am3.saveModel model="${model.name}" path="${metamodel.path}${model.name}.xmi">
            </am3.saveModel>
      </target>
</project>
```

Figure 5.8: AM3 Ant task which transforms OCL code into the OCL model

The result of parsing the OCL code (EBNF injection from Figure 5.5) is an OCL model shown in Figure 5.9 (a). The model is shown by using UML object diagrams in order to keep the representation both simple and descriptive enough. This OCL model is serialized by the NetBeans MetaData Repository (MDR) model handler into the OCL XMI format (MOF-1.4 based), which is shown in Figure 5.9(b). When OCL code is parsed, or when an OCL model is transformed, it is possible to choose model handler (MDR or EMF).



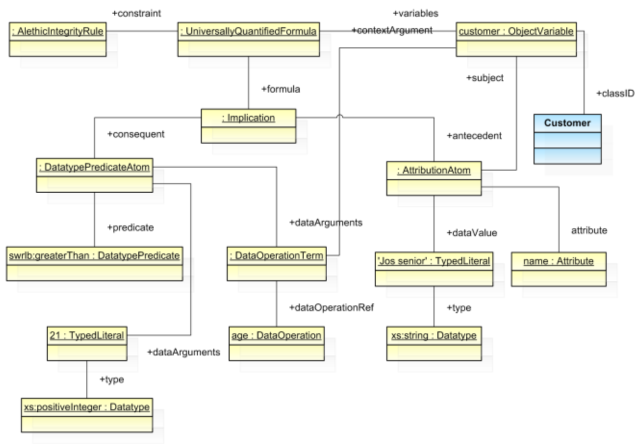
```
<OCL.OclModule xmi.id = 'a1'>
  <OCL.OclModule.ownedElements>
    <OCL.Invariant xmi.id = 'a2'>
      <OCL.OclModuleElement.contextDefinition>
        <OCL.OclContextDefinition xmi.id = 'a3'>
          <OCL.OclContextDefinition.contextElement>
            <OCL.Class xmi.id = 'a4' name = 'Customer'
                       isAbstract = 'false'/>
          </OCL.OclContextDefinition.contextElement>
        </OCL.OclContextDefinition>
      </OCL.OclModuleElement.contextDefinition>
      <OCL.Invariant.specification>
        <OCL.OperatorCallExp xmi.id = 'a5' name = 'implies'>
          <OCL.CallExp.source>
            <OCL.OperatorCallExp xmi.id = 'a6' name = '='>
              <OCL.CallExp.source>
                <OCL.PropertyCallExp xmi.id = 'a7' name = 'name'>
                  <OCL.CallExp.source>
                    <OCL.VariableExp xmi.id = 'a8'>
                      <OCL.VariableExp.referredVariable>
                        <OCL.Variable xmi.idref = 'a9'/>
                      </OCL.VariableExp.referredVariable>
                    </OCL.VariableExp>
                  </OCL.CallExp.source>
                </OCL.PropertyCallExp>
              </OCL.CallExp.source>

          <!-- ... --->

        </OCL.OperatorCallExp>
       </OCL.Invariant.specification>
     </OCL.Invariant>
  </OCL.OclModule.ownedElements>
</OCL.OclModule>
```

a) UML object diagram of the OCL model attained from the rule which shown in Figure 5.7

b) OCL XMI representation of the rule shown in Figure 5.7

Figure 5.9: OCL model and OCL XMI representation of the OCL rule shown in Figure 5.7

The MOF-based OCL model (serialized in the OCL XMI format) can be used as an input for ATL transformation. The OCL model shown in Figure 5.9 is transformed to an R2ML model (see Figure 5.10), by using the ATL transformation OCL2R2ML.atl (shown in Figure 5.5). This transformation is based on conceptual mappings between OCL and R2ML meta-models (Table 4.8.) and is described in details in section 4.4.2, and Figure 4.2 (transformation 8. OCL2R2ML).



```
<R2ML.RuleBase xmi.id = 'a1' ruleBaseID = 'OCL'>
  <R2ML.RuleBase.rules>
    <R2ML.IntegrityRuleSet xmi.id = 'a2'>
      <R2ML.IntegrityRuleSet.rules>
        <R2ML.Rules.AlethicIntegrityRule xmi.id = 'a3' ruleID = ''>
          <R2ML.Rules.IntegrityRule.constraint>
            <R2ML.Formulas.UniversallyQuantifiedFormula xmi.id = 'a4'>
              <R2ML.Formulas.QuantifiedFormula.variables>
                 <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a5'/>
              </R2ML.Formulas.QuantifiedFormula.variables>
              <R2ML.Formulas.QuantifiedFormula.formula>
                <R2ML.Formulas.Implication xmi.id = 'a9'>
                  <R2ML.Formulas.Implication.consequent>
                     <R2ML.Atoms.AtBasic.DatatypePredicateAtom xmi.id = 'a10'
                      isNegated = 'false'>
                       <R2ML.Atoms.AtBasic.DatatypePredicateAtom.dataArguments>
                          <R2ML.Terms.TerFunctional.DataOperationTerm xmi.idref = 'a7'/>
                          <R2ML.Vocabulary.VocBasic.TypedLiteral xmi.idref = 'a11'/>
                       </R2ML.Atoms.AtBasic.DatatypePredicateAtom.dataArguments>
                       <R2ML.Atoms.AtBasic.DatatypePredicateAtom.predicate>
                          <R2ML.Vocabulary.VocBasic.DatatypePredicate xmi.idref = 'a12'/>
                       </R2ML.Atoms.AtBasic.DatatypePredicateAtom.predicate>
                     </R2ML.Atoms.AtBasic.DatatypePredicateAtom>
                  </R2ML.Formulas.Implication.consequent>
                </R2ML.Formulas.Implication>
              </R2ML.Formulas.QuantifiedFormula.formula>
            </R2ML.Formulas.UniversallyQuantifiedFormula>

            <!-- ... -->

           </R2ML.Rules.IntegrityRule.constraint>
         </R2ML.Rules.AlethicIntegrityRule>
       </R2ML.IntegrityRuleSet.rules>
     </R2ML.IntegrityRuleSet>
  </R2ML.RuleBase.rules>
</R2ML.RuleBase>
```

a) UML object diagram of the R2ML model transformed from the OCL rule shown in Figure 5.7

b) R2ML XMI representation of the transformed OCL rule shown in Figure 5.7

Figure 5.10: R2ML model and R2ML XMI representation of the transformed OCL rule from Figure 5.7

In figures 5.9 and 5.10 one can see that OCL elements are transformed to corresponding R2ML element, conforming to the conceptual mappings between these two languages presented in Table 4.8 (section 4.4). Also, one can notice that the OCL `OclModule` element from Figure 5.9, is transformed into the R2ML `AlethicIntegrityRule` element from Figure 5.10. In addition, OCL `OperatorCallExp` (with the name attribute "implies") is transformed into the R2ML `Implication` element. Operator "=" from OCL, is represented with `OperatorCallExp` in OCL model, and it is transformed to R2ML `AttributionAtom`. While the OCL operator ">", which is also represented with `OperatorCallExp` in OCL model, is transformed into the `DatatypePredicateAtom` R2ML element. Contextual `Class` OCL element (with name *Customer*) is directly transformed into the R2ML `Class` element (with the same name).

The R2ML model shown in Figure 5.10, is serialized into the R2ML rule in R2ML concrete syntax by using transformation R2ML2XML.atl (shown in Figure 5.5). In particular, we first get an XML model (Figure 5.11) which is subsequently transformed into the XML technological space (Figure 5.12) by using the built-in ATL XML extractor. This transformation process of a R2ML rule from abstract syntax into the R2ML concrete syntax is described in section 4.2.1.2 and shown in Figure 4.2 (transformation 2. R2ML2XML).

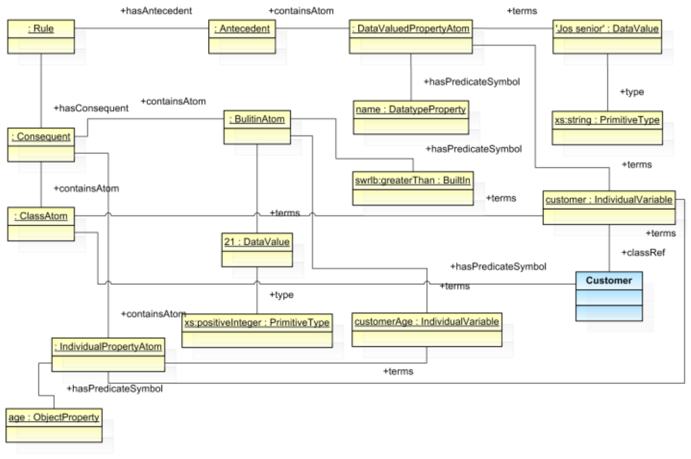
```
<XML.Root xmi.id = 'a1' name = 'r2ml:RuleBase'>
  <XML.Element.children>
    <XML.Element xmi.id = 'a6' name = 'r2ml:IntegrityRuleSet'>
      <XML.Element.children>
        <XML.Element xmi.id = 'a7' name = 'r2ml:AlethicIntegrityRule'>
          <XML.Element.children>
             <XML.Attribute xmi.id = 'a8' name = 'r2ml:ruleID' value = ''/>
             <XML.Element xmi.id = 'a9' name = 'r2ml:constraint'>
               <XML.Element.children>
                 <XML.Element xmi.id = 'a10' name = 'r2ml:UniversallyQuantifiedFormula'>
                   <XML.Element.children>
                     <XML.Element xmi.id = 'a11' name = 'r2ml:ObjectVariable'>
                       <XML.Element.children>
                         <XML.Attribute xmi.id = 'a12' name = 'r2ml:name' value = 'customer'/>
                         <XML.Attribute xmi.id = 'a13' name = 'r2ml:classID' value = 'Customer'/>
                       </XML.Element.children>
                     </XML.Element>
                     <XML.Element xmi.id = 'a14' name = 'r2ml:Implication'>
                       <XML.Element.children>
                         <XML.Element xmi.id = 'a15' name = 'r2ml:antecedent'>
                           <XML.Element.children>
                             <XML.Element xmi.id = 'a16' name = 'r2ml:AttributionAtom'>
                               <XML.Element.children>
                                 <XML.Attribute xmi.id = 'a17' name = 'r2ml:attributeID'
                                                value = 'name'/>
                                 <XML.Element xmi.id = 'a18' name = 'r2ml:subject'>
                                   <XML.Element.children>
                                     <XML.Element xmi.id = 'a19' name = 'r2ml:ObjectVariable'>
                                       <XML.Element.children>
                                         <XML.Attribute xmi.id = 'a20' name = 'r2ml:name'
                                                        value = 'customer'/>
                                         <XML.Attribute xmi.id = 'a21' name = 'r2ml:classID'
                                                        value = 'Customer'/>
                                       </XML.Element.children>
                                     </XML.Element>
                                   </XML.Element.children>
                                 </XML.Element>
                                 <!-- ... -->
                               </XML.Element.children>
                             </XML.Element>
                           </XML.Element.children>
                         </XML.Element>
                        <!-- ... -->
                     </XML.Element>
                   </XML.Element.children>
                 </XML.Element>
               </XML.Element.children>
             </XML.Element>
          </XML.Element.children>
        </XML.Element>
      </XML.Element.children>
    </XML.Element>
  </XML.Element.children>
</XML.Root>
```

Figure 5.11: XML model in XML XMI format transformed from R2ML model which is shown in Figure 5.10

```
<r2ml:RuleBase xmlns:dc = 'http://purl.org/dc/elements/1.1/'
               xsi:schemaLocation = 'http://www.rewerse.net/I1/2006/R2ML
                                     http://oxygen.informatik.tu-cottbus.de/R2ML/0.4/R2ML.xsd'
               xmlns:r2ml = 'http://www.rewerse.net/I1/2006/R2ML'
               xmlns:xsi = 'http://www.w3.org/2001/XMLSchema-instance'>
 <r2ml:IntegrityRuleSet>
   <r2ml:AlethicIntegrityRule r2ml:ruleID = ''>
     <r2ml:constraint>
       <r2ml:UniversallyQuantifiedFormula>
         <r2ml:ObjectVariable r2ml:name = 'customer' r2ml:classID = 'Customer'/>
         <r2ml:Implication>
           <r2ml:antecedent>
             <r2ml:AttributionAtom r2ml:attributeID = 'name'>
               <r2ml:subject>
                 <r2ml:ObjectVariable r2ml:name = 'customer'
                                      r2ml:classID = 'Customer'/>
               </r2ml:subject>
               <r2ml:dataValue>
                 <r2ml:TypedLiteral r2ml:datatypeID = 'xs:string'
                                    r2ml:lexicalValue = 'Jos senior'/>
               </r2ml:dataValue>
             </r2ml:AttributionAtom>
           </r2ml:antecedent>
           <r2ml:consequent>
             <r2ml:DatatypePredicateAtom r2ml:datatypePredicateID = 'swrlb:greaterThan'>
               <r2ml:dataArguments>
                 <r2ml:DataOperationTerm r2ml:operationID = 'age'>
                   <r2ml:contextArgument>
                     <r2ml:ObjectVariable r2ml:name = 'customer'
                                          r2ml:classID = 'Customer'/>
                   </r2ml:contextArgument>
                 </r2ml:DataOperationTerm>
                 <r2ml:TypedLiteral r2ml:datatypeID = 'xs:positiveInteger'
                                    r2ml:lexicalValue = '21'/>
               </r2ml:dataArguments>
             </r2ml:DatatypePredicateAtom>
           </r2ml:consequent>
         </r2ml:Implication>
       </r2ml:UniversallyQuantifiedFormula>
     </r2ml:constraint>
   </r2ml:AlethicIntegrityRule>
 </r2ml:IntegrityRuleSet>
```

Figure 5.12: R2ML XML representation of the R2ML model shown in Figure 5.10

When the R2ML rule from Figure 5.10 is located in a model repository, or it exists in concrete XML-based syntax, it can be transformed into an RDM model (as instance of RDM meta-model, which represents SWRL abstract syntax). The R2ML model is transformed into an RDM model by R2ML2RDM.atl transformation, which is shown in Figure 5.5, while mappings between these meta-model elements conform to those from Table 4.6. The transformation process of the R2ML model into the RDM model is described in section 4.3.2 and shown in Figure 4.2 (transformation 6. R2ML2RDM). The result of this transformation is the RDM model shown in Figure 5.13.



```
<RDM.Rule xmi.id = 'a4'>
  <RDM.Rule.hasAntecedent>
    <RDM.Antecedent xmi.idref = 'a5'/>
  </RDM.Rule.hasAntecedent>
  <RDM.Rule.hasConsequent>
    <RDM.Consequent xmi.idref = 'a6'/>
  </RDM.Rule.hasConsequent>
</RDM.Rule>
<RDM.BuiltIn xmi.id = 'a7' buildInID = 'swrlb:greaterThan'/>
<RDM.Antecedent xmi.id = 'a5'>
  <RDM.Antecedent.containsAtom>
    <RDM.Atom xmi.idref = 'a8'/>
  </RDM.Antecedent.containsAtom>
</RDM.Antecedent>
<!-- ... -->
<RDM.Atom xmi.id = 'a8' name = 'DataValuedPropertyAtom'>
  <RDM.Atom.terms>
    <RDM.ODM.DataValue xmi.idref = 'a9'/>
    <RDM.IndividualVariable xmi.idref = 'a1'/>
  </RDM.Atom.terms>
  <RDM.Atom.hasPredicateSymbol>
    <RDM.ODM.DatatypeProperty xmi.idref = 'a10'/>
  </RDM.Atom.hasPredicateSymbol>
</RDM.Atom>
```

a) UML object diagram of RDM model transformed from the R2ML rule shown in Figure 5.12

b) RDM XMI representation of the transformed R2ML rule shown in Figure 5.12

Figure 5.13: RDM model and RDM XMI representation of the transformed R2ML rule shown in Figure 5.12

R2ML `AlethicIntegrityRule` element (Figure 5.10) is transformed into the RDM `Rule` element, (Figure 5.13). The `DatatypePredicateAtom` R2ML element with `DatatypePredicate` (*swrlb:greaterThan*) is transformed into the RDM `BuiltInAtom` with the `BuiltIn` element (*swrlb:greaterThan*). R2ML `AttributionAtom` with the `Attribute` element (*name*) is transformed into RDM `DataValuedPropertyAtom` with `DataTypeProperty` (*name*).

In order to transform the RDM model obtained in the last step, into the SWRL XML concrete syntax, we first need to transform it into an XML model (see Figure 5.14a), by using transformation RDM2XML.atl, which is shown in Figure 5.5. The XML model is serialized into the XML technological space by using XML extractor (XML extraction in Figure 5.5). The transformation process of a rule represented in the SWRL abstract syntax (i.e., RDM meta-model) into rule represented in the SWRL XML concrete syntax is described in section 4.2.2.2 and shown in Figure 4.2 (transformation 4. RDM2XML). The result of this action is the SWRL rule shown in Figure 5.14b.

```
<XML.Root xmi.id = 'a1' name = 'ruleml:imp'>
     <XML.Element.children>
       <XML.Element xmi.id = 'a5' name = 'ruleml:_body'>
         <XML.Element.children>
           <XML.Element xmi.id = 'a6' name = 'swrlx:datavaluedPropertyAtom'>
             <XML.Element.children>
               <XML.Attribute xmi.id = 'a7' name = 'swrlx:property'
                                            value = 'name'/>
               <XML.Element xmi.id = 'a8' name = 'ruleml:var'>
                 <XML.Element.children>
                   <XML.Text xmi.id = 'a9' name = '#text' value = 'customer'/>
                 </XML.Element.children>
               </XML.Element>
               <XML.Element xmi.id = 'a10' name = 'owlx:DataValue'>
                 <XML.Element.children>
                   <XML.Attribute xmi.id = 'a11' name = 'owlx:datatype'
                                                 value = 'xs:string'/>
                   <XML.Text xmi.id = 'a12' name = '#text'
                                            value = 'Jos senior'/>
                 </XML.Element.children>
               </XML.Element>
             </XML.Element.children>
           </XML.Element>
         </XML.Element.children>
       </XML.Element>
      <!-- ... -->
     </XML.Element.children>
</XML.Root>
```

a) XML XMI representation of the transformed RDM model shown in Figure 5.13

```
<ruleml:Implies>
 <ruleml:body>
   <swrlx:datavaluedPropertyAtom swrlx:property = 'name'>
     <ruleml:var>customer</ruleml:var>
     <owlx:DataValue owlx:datatype = 'xs:string'>Jos senior</owlx:DataValue>
   </swrlx:datavaluedPropertyAtom>
 </ruleml:body>
 <ruleml:head>
   <swrlx:classAtom>
     <owlx:Class owlx:name = 'Customer'/>
     <ruleml:var>customer</ruleml:var>
   </swrlx:classAtom>
   <swrlx:individualPropertyAtom swrlx:property = 'age'>
     <ruleml:var>customer</ruleml:var>
     <ruleml:var>customerAge</ruleml:var>
   </swrlx:individualPropertyAtom>
   <swrlx:builtinAtom swrlx:builtin = 'swrlb:greaterThan'>
     <ruleml:var>customerAge</ruleml:var>
     <owlx:DataValue owlx:datatype = 'xs:positiveInteger'>21</owlx:DataValue>
   </swrlx:builtinAtom>
 </ruleml:head>
</ruleml:Implies>
```

b) SWRL rule of the transformed RDM model shown in Figure 5.13

Figure 5.14: XML model and SWRL rule of the transformed RDM model shown in Figure 5.13

The RDM model elements are transformed into the XML model elements (that conforms to SWRL/OWL XML Schema) by using RDM2XML.atl transformation, which is shown in Figure 5.5. The SWRL rule from Figure 5.13 (b) defines implication which in its head part has a `classAtom` with "customer" variable of the type `Customer`, an `individualPropertyAtom` with "*customerAge*" variable which represents the "*age*" property of the "*customer*" variable, and a `builtinAtom` that compares the "*customerAge*" variable with the value 21 (`DataValue`) by using builtin with name "*swrlb:greaterThan*". In the body part of the SWRL rule there is a `datavaluedPropertyAtom` that assigns string "*Jos senior*" to the "*name*" property of the "*customer*" variable.

5.2.1.2. SWRL to OCL

The transformation of SWRL rules compliant with the SWRL (OWL) XML Schema, into the OCL invariants, via R2ML as the intermediary rule language, is based on the transformation scenario shown in Figure 5.4. Figure 5.15 shows a SWRL rule in concrete XML-based syntax and represents how built-ins can be applied to perform comparisons necessary to compute a discount. This rule is contained in an ontology (with the name "Ontology") and it is defined as: *Gold customers get a 10% discount on purchases of $500 or more*.

```
<swrlx:Ontology swrlx:name = "Ontology">
  <ruleml:Implies>
      <owlx:Annotation>
           <owlx:Documentation>Gold customers get a 10% discount
                               on purchases of $500 or more
           </owlx:Documentation>
      </owlx:Annotation>
      <ruleml:body>
         <swrlx:individualPropertyAtom swrlx:property="hasStatus">
            <ruleml:var>customer</ruleml:var>
            <owlx:Individual owlx:name="gold"/>
         </swrlx:individualPropertyAtom>
         <swrlx:datavaluedPropertyAtom swrlx:property="hasTotalPurchase">
            <ruleml:var>customer</ruleml:var>
            <ruleml:var>total</ruleml:var>
         </swrlx:datavaluedPropertyAtom>
         <swrlx:builtinAtom swrlx:builtin="greaterThanOrEqual">
            <ruleml:var>total</ruleml:var>
            <owlx:DataValue owlx:datatype="int">500</owlx:DataValue>
         </swrlx:builtinAtom>
      </ruleml:body>
      <ruleml:head>
         <swrlx:datavaluedPropertyAtom swrlx:property="hasDiscount">
            <ruleml:var>customer</ruleml:var>
           <owlx:DataValue owlx:datatype="int">10</owlx:DataValue>
         </swrlx:datavaluedPropertyAtom>
      </ruleml:head>
   </ruleml:Implies>

   <owlx:ObjectProperty owlx:name="hasStatus">
       <owlx:domain>
               <owlx:Class owlx:name="Customer"/>
       </owlx:domain>
       <owlx:range>
               <owlx:Individual owlx:name="gold"/>
       </owlx:range>
  </owlx:ObjectProperty>
</swrlx:Ontology>
```

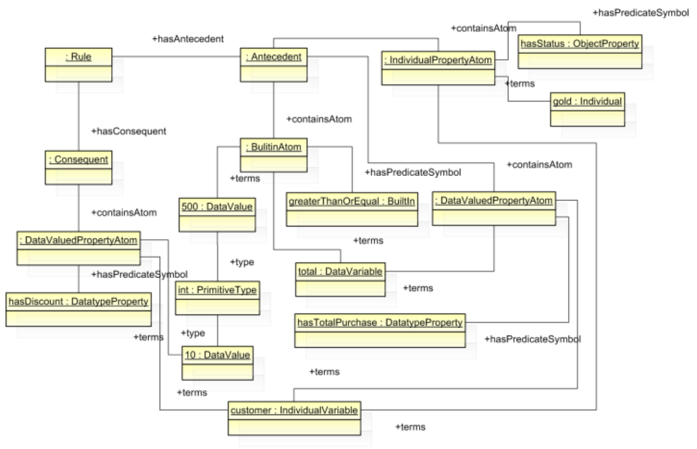
Figure 5.15: A SWRL rule in XML-based concrete syntax

The SWRL rule, shown in Figure 5.15 is transformed from the XML technological space into the MOF technological space by using ATL's XML injector (XML injection is represented in Figure 5.5). Subsequently, this rule can be exported into the XML XMI format by using NetBeans Metadata Repository (MDR), as it is shown in Figure 5.16. The process of transforming a SWRL rule from the SWRL XML concrete syntax into the SWRL abstract syntax (i.e., RDM meta-model) is described in section 4.2.2.1 and shown in Figure 4.2 (transformation 3. XML2RDM).

```
<XML.Root xmi.id = 'a1' name = 'ruleml:imp' value = ''>
     <XML.Element.children>
       <XML.Element xmi.id = 'a2' name = 'ruleml:_body' value = ''>
         <XML.Element.children>
           <XML.Element xmi.id = 'a3' name = 'swrlx:builtinAtom' value = ''>
             <XML.Element.children>
               <XML.Attribute xmi.id = 'a4' name = 'swrlx:builtin' value = 'greaterThanOrEqual'/>
               <XML.Element xmi.id = 'a5' name = 'owlx:DataValue' value = ''>
                 <XML.Element.children>
                   <XML.Attribute xmi.id = 'a6' name = 'owlx:datatype' value = 'int'/>
                   <XML.Text xmi.id = 'a7' name = '#text' value = '500'/>
                 </XML.Element.children>
               </XML.Element>
               <XML.Element xmi.id = 'a8' name = 'ruleml:var' value = ''>
                 <XML.Element.children>
                   <XML.Text xmi.id = 'a9' name = '#text' value = 'total'/>
                 </XML.Element.children>
               </XML.Element>
             </XML.Element.children>
           </XML.Element>
           <!-- ... -->
         </XML.Element.children>
       </XML.Element>
    </XML.Element.children>
</XML.Root>
```

Figure 5.16: XML XMI representation of the SWRL rule shown in Figure 5.15

The XML model shown in Figure 5.16 can be transformed to the corresponding RDM model, by using the XML2RDM.atl transformation shown in Figure 5.5. The result of executing this transformation on the XML model shown in Figure 5.16 is the RDM model shown in Figure 5.17.



```
<RDM.Rule xmi.id = 'a2'>
  <RDM.Rule.hasAntecedent>
     <RDM.Antecedent xmi.idref = 'a3'/>
  </RDM.Rule.hasAntecedent>
  <RDM.Rule.hasConsequent>
     <RDM.Consequent xmi.idref = 'a4'/>
  </RDM.Rule.hasConsequent>
</RDM.Rule>
<RDM.Antecedent xmi.id = 'a3'>
  <RDM.Antecedent.containsAtom>
     <RDM.Atom xmi.idref = 'a7'/>
     <RDM.Atom xmi.idref = 'a8'/>
     <RDM.Atom xmi.idref = 'a9'/>
  </RDM.Antecedent.containsAtom>
</RDM.Antecedent>
<!--...-->
<RDM.Atom xmi.id = 'a9' name = 'IndividualPropertyAtom'>
  <RDM.Atom.terms>
     <RDM.IndividualVariable xmi.idref = 'a1'/>
     <RDM.ODM.Individual xmi.idref = 'a10'/>
  </RDM.Atom.terms>
  <RDM.Atom.hasPredicateSymbol>
     <RDM.ODM.ObjectProperty xmi.idref = 'a11'/>
  </RDM.Atom.hasPredicateSymbol>
</RDM.Atom>
```

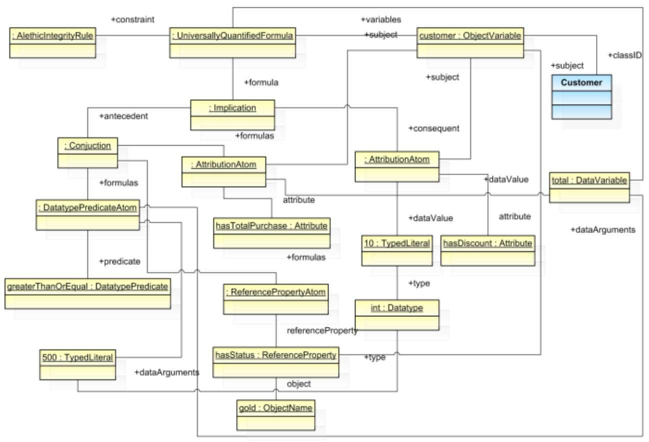
a) UML object diagram of RDM model transformed from the XML model which is shown in Figure 5.16

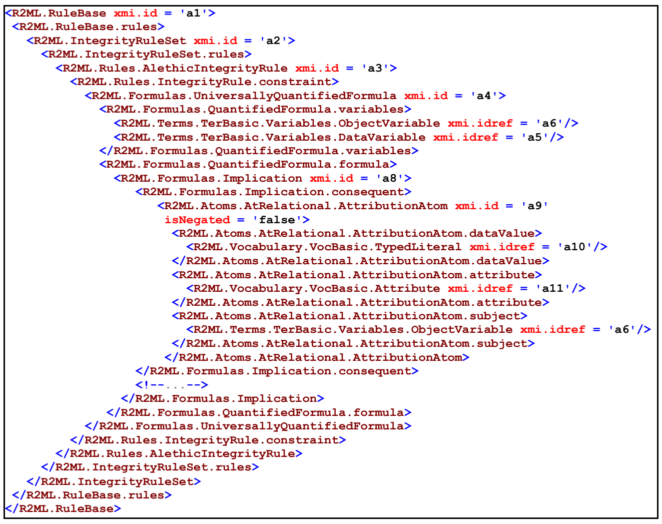
b) RDM XMI representation of the transformed XML model which is shown in Figure 5.16

Figure 5.17: RDM model and RDM XMI representation of the transformed XML model from Figure 5.16

In this transformation, the XML model elements shown in Figure 5.16, are transformed to the equivalent RDM model elements (it also includes ODM model elements). This transformation is shown in Figure 5.17. The XML `Root` element with the name attribute '*ruleml:imp*' is transformed into the RDM `Rule` element. Other elements are transformed in a similar way, as it is shown in the mappings given in Table 4.6.

The RDM model shown in Figure 5.17 is transformed into the analoguous R2ML model by using the RDM2R2ML.atl transformation (shown in Figure 5.5), which conforms to the mappings from Table 4.6. The transformation process between the RDM model and the R2ML model is described in section 4.3.1 and shown in Figure 4.2 (transformation 5. RDM2R2ML). The execution of this transformation on the RDM model shown in Figure 5.17, results in the R2ML model shown in Figure 5.18.





```
<R2ML.RuleBase xmi.id = 'a1'>
 <R2ML.RuleBase.rules>
   <R2ML.IntegrityRuleSet xmi.id = 'a2'>
     <R2ML.IntegrityRuleSet.rules>
       <R2ML.Rules.AlethicIntegrityRule xmi.id = 'a3'>
         <R2ML.Rules.IntegrityRule.constraint>
           <R2ML.Formulas.UniversallyQuantifiedFormula xmi.id = 'a4'>
             <R2ML.Formulas.QuantifiedFormula.variables>
               <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a6'/>
               <R2ML.Terms.TerBasic.Variables.DataVariable xmi.idref = 'a5'/>
             </R2ML.Formulas.QuantifiedFormula.variables>
             <R2ML.Formulas.QuantifiedFormula.formula>
               <R2ML.Formulas.Implication xmi.id = 'a8'>
                  <R2ML.Formulas.Implication.consequent>
                     <R2ML.Atoms.AtRelational.AttributionAtom xmi.id = 'a9'
                      isNegated = 'false'>
                       <R2ML.Atoms.AtRelational.AttributionAtom.dataValue>
                         <R2ML.Vocabulary.VocBasic.TypedLiteral xmi.idref = 'a10'/>
                       </R2ML.Atoms.AtRelational.AttributionAtom.dataValue>
                       <R2ML.Atoms.AtRelational.AttributionAtom.attribute>
                         <R2ML.Vocabulary.VocBasic.Attribute xmi.idref = 'a11'/>
                       </R2ML.Atoms.AtRelational.AttributionAtom.attribute>
                       <R2ML.Atoms.AtRelational.AttributionAtom.subject>
                         <R2ML.Terms.TerBasic.Variables.ObjectVariable xmi.idref = 'a6'/>
                       </R2ML.Atoms.AtRelational.AttributionAtom.subject>
                      </R2ML.Atoms.AtRelational.AttributionAtom>
                  </R2ML.Formulas.Implication.consequent>
                  <!--...-->
                </R2ML.Formulas.Implication>
              </R2ML.Formulas.QuantifiedFormula.formula>
           </R2ML.Formulas.UniversallyQuantifiedFormula>
         </R2ML.Rules.IntegrityRule.constraint>
       </R2ML.Rules.AlethicIntegrityRule>
     </R2ML.IntegrityRuleSet.rules>
   </R2ML.IntegrityRuleSet>
 </R2ML.RuleBase.rules>
</R2ML.RuleBase>
```

a) UML object diagram of R2ML model transformed from the RDM model which is shown in Figure 5.17

b) R2ML XMI representation of the transformed RDM model which is shown in Figure 5.17

Figure 5.18: R2ML model and R2ML XMI representation of the transformed RDM model shown in Figure 5.17

In this transformation, the RDM Rule element shown in Figure 5.17 is transformed into the R2ML `AlethicIntegrityRule` element shown in Figure 5.18, with the `UniversallyQuantifiedFormula` element as constraint and the `Implication` element as formula. RDM `DataValuedPropertyAtom` with `DatatypeProperty` (*hasDiscount*) is transformed into the corresponding `AttributionAtom` with an `Attribute` element (*hasDiscount*). `DataValuedPropertyAtom` with `DatatypeProperty` (*hasTotalPurchase*) is transformed into the equivalent `AttributionAtom` with an `Attribute` element (*hasTotalPurchase*). RDM `IndividualPropertyAtom` is transformed into an R2ML `ReferencePropertyAtom`, whereas `ObjectProperty` is transformed into a `ReferenceProperty`. RDM `BuiltinAtom` with the "greaterThanOrEqual" `BuiltIn` element, is transformed into the `DatatypePredicateAtom` with a `DatatypePredicate` element (*greaterThanOrEqual*). Finally, the RDM element `Individual` (*gold*) is transformed into the R2ML `ObjectName` element (*gold*).

The R2ML model shown in Figure 5.18 is then serialized into the corresponding R2ML rule in the R2ML XML concrete syntax. By using the R2ML2XML.atl transformation (shown in Figure 5.5), an R2ML model transforms into the XML model (see Figure 5.19), which is serialized with ATL extractor into the R2ML XML concrete syntax (see Figure 5.2) conformant to R2ML XML Schema. The transformation process of an R2ML rule from the abstract syntax into the R2ML XML concrete syntax is described in section 4.2.1.2 and shown in Figure 4.2 (transformation 2. R2ML2XML).

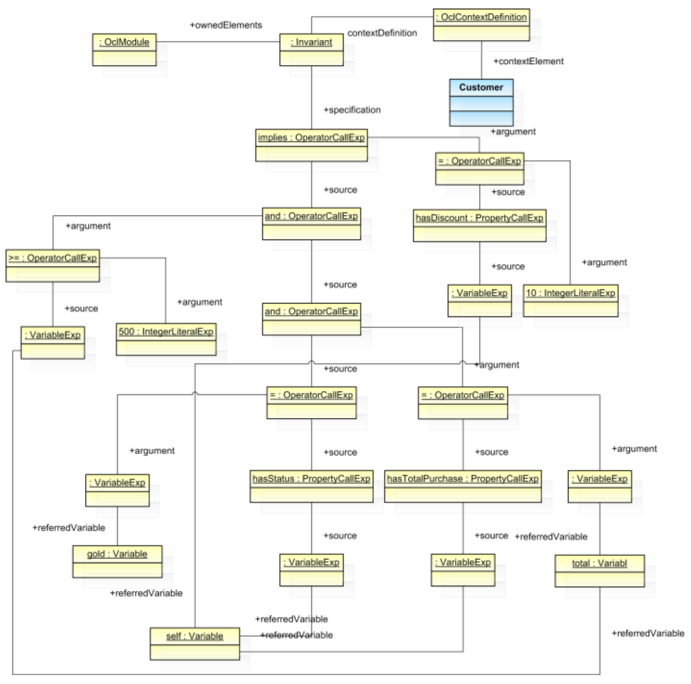
```
<XML.Root xmi.id = 'a1' name = 'r2ml:RuleBase'>
   <XML.Element.children>
       <XML.Element xmi.id = 'a6' name = 'r2ml:IntegrityRuleSet'>
         <XML.Element.children>
           <XML.Element xmi.id = 'a7' name = 'r2ml:AlethicIntegrityRule'>
             <XML.Element.children>
               <XML.Element xmi.id = 'a9' name = 'r2ml:constraint'>
                 <XML.Element.children>
                   <XML.Element xmi.id = 'a10' name = 'r2ml:UniversallyQuantifiedFormula'>
                     <XML.Element.children>
                       <XML.Element xmi.id = 'a11' name = 'r2ml:ObjectVariable'>
                         <XML.Element.children>
                           <XML.Attribute xmi.id = 'a12' name = 'r2ml:name' value = 'customer'/>
                         </XML.Element.children>
                       </XML.Element>
                       <XML.Element xmi.id = 'a13' name = 'r2ml:DataVariable'>
                         <XML.Element.children>
                           <XML.Attribute xmi.id = 'a14' name = 'r2ml:name' value = 'total'/>
                         </XML.Element.children>
                       </XML.Element>
                       <XML.Element xmi.id = 'a16' name = 'r2ml:Implication'>
                         <XML.Element.children>
                           <XML.Element xmi.id = 'a17' name = 'r2ml:antecedent'>
                             <XML.Element.children>
                               <XML.Element xmi.id = 'a18' name = 'r2ml:Conjunction'>
                                 <XML.Element.children>
                                   <XML.Element xmi.id = 'a19' name = 'r2ml:DatatypePredicateAtom'>
                                     <XML.Element.children>
                                       <XML.Attribute xmi.id = 'a20' name = 'r2ml:datatypePredicateID'
                                        value = 'greaterThanOrEqual'/>
                                       <XML.Element xmi.id = 'a21' name = 'r2ml:dataArguments'>
                                         <XML.Element.children>
                                           <XML.Element xmi.id = 'a22' name = 'r2ml:DataVariable'>
                                             <XML.Element.children>
                                               <XML.Attribute xmi.id = 'a23' name = 'r2ml:name'
                                                value = 'total'/>
                                             </XML.Element.children>
                                           </XML.Element>
                                         <!-- ... -->
                                         </XML.Element.children>
                                       </XML.Element>
                                     </XML.Element.children>
                                   </XML.Element>
                                 </XML.Element.children>
                               </XML.Element>
                             </XML.Element.children>
                           </XML.Element>
                           <!-- ... -->
                       </XML.Element>
                     </XML.Element.children>
                   </XML.Element>
                 </XML.Element.children>
               </XML.Element>
             </XML.Element.children>
           </XML.Element>
         </XML.Element.children>
       </XML.Element>
   </XML.Element.children>
</XML.Root>
```

Figure 5.19: XML model in XML XMI format transformed from the R2ML model shown in Figure 5.18

```
<r2ml:AlethicIntegrityRule>
  <r2ml:constraint>
     <r2ml:UniversallyQuantifiedFormula>
       <r2ml:ObjectVariable r2ml:name = 'customer'/>
       <r2ml:DataVariable r2ml:name = 'total' r2ml:typeCategory = 'individual'/>
       <r2ml:Implication>
         <r2ml:antecedent>
           <r2ml:Conjunction>
             <r2ml:DatatypePredicateAtom r2ml:datatypePredicateID = 'greaterThanOrEqual'>
               <r2ml:dataArguments>
                 <r2ml:DataVariable r2ml:name = 'total' r2ml:typeCategory = 'individual'/>
                 <r2ml:TypedLiteral r2ml:datatypeID = 'int' r2ml:lexicalValue = '500'/>
               </r2ml:dataArguments>
             </r2ml:DatatypePredicateAtom>
             <r2ml:AttributionAtom r2ml:attributeID = 'hasTotalPurchase'>
               <r2ml:subject>
                 <r2ml:ObjectVariable r2ml:name = 'customer'/>
               </r2ml:subject>
               <r2ml:dataValue>
                 <r2ml:DataVariable r2ml:name = 'total' r2ml:typeCategory = 'individual'/>
               </r2ml:dataValue>
             </r2ml:AttributionAtom>
             <r2ml:ReferencePropertyAtom r2ml:referencePropertyID = 'hasStatus'>
               <r2ml:subject>
                 <r2ml:ObjectVariable r2ml:name = 'customer'/>
               </r2ml:subject>
               <r2ml:object>
                 <r2ml:ObjectName r2ml:objectID = 'gold'/>
               </r2ml:object>
             </r2ml:ReferencePropertyAtom>
           </r2ml:Conjunction>
         </r2ml:antecedent>
         <r2ml:consequent>
           <r2ml:AttributionAtom r2ml:attributeID = 'hasDiscount'>
             <r2ml:subject>
               <r2ml:ObjectVariable r2ml:name = 'customer'/>
             </r2ml:subject>
             <r2ml:dataValue>
               <r2ml:TypedLiteral r2ml:datatypeID = 'int' r2ml:lexicalValue = '10'/>
             </r2ml:dataValue>
           </r2ml:AttributionAtom>
         </r2ml:consequent>
       </r2ml:Implication>
     </r2ml:UniversallyQuantifiedFormula>
 </r2ml:constraint>
</r2ml:AlethicIntegrityRule>
```

Figure 5.20: R2ML XML representation of R2ML model which is shown in Figure 5.18

The R2ML model shown in Figure 5.18, can be either serialized in XMI format (Figure 5.18b) or can be used as an input for ATL transformation. In the transformation scenario (see Figure 5.4) between SWRL and OCL, the next step is to transform the attained R2ML model into the OCL model, by using the R2ML2OCL.atl transformation, which is shown in Figure 5.5. After the execution of this transformation we get an OCL model, which is shown in Figure 5.21. This transformation process of the R2ML meta-model into the OCL meta-model is described in section 4.4.1 and shown in Figure 4.2 (transformation 7. R2ML2OCL).



```
<OCL.OclModule xmi.id = 'a1'>
  <OCL.OclModule.ownedElements>
    <OCL.Invariant xmi.id = 'a2'>
      <OCL.OclModuleElement.contextDefinition>
        <OCL.OclContextDefinition xmi.id = 'a3'>
          <OCL.OclContextDefinition.contextElement>
            <OCL.Class xmi.id = 'a4' name = 'Customer'/>
          </OCL.OclContextDefinition.contextElement>
        </OCL.OclContextDefinition>
      </OCL.OclModuleElement.contextDefinition>
      <OCL.Invariant.specification>
       <OCL.OperatorCallExp xmi.id = 'a5' name = 'implies'>
         <OCL.CallExp.source>
           <OCL.OperatorCallExp xmi.id = 'a6' name = 'and'>
             <OCL.CallExp.source>
               <OCL.OperatorCallExp xmi.id = 'a7' name = 'and'>
                 <OCL.CallExp.source>
                   <OCL.OperatorCallExp xmi.id = 'a8' name = '='>
                     <OCL.CallExp.source>
                       <OCL.PropertyCallExp xmi.id = 'a9'
                        name = 'hasStatus'>
                         <OCL.CallExp.source>
                           <OCL.VariableExp xmi.id = 'a10'>
                             <OCL.VariableExp.referredVariable>
                               <OCL.Variable xmi.idref = 'a11'/>
                             </OCL.VariableExp.referredVariable>
                           </OCL.VariableExp>
                         </OCL.CallExp.source>
                       </OCL.PropertyCallExp>
                     </OCL.CallExp.source>
                     <!--...-->
                   </OCL.OperatorCallExp>
                 </OCL.CallExp.source>
                 <!--...-->
               </OCL.OperatorCallExp>
             </OCL.CallExp.source>
             <!--...-->
           </OCL.OperatorCallExp>
         </OCL.CallExp.source>
         <!--...-->
       </OCL.OperatorCallExp>
      </OCL.Invariant.specification>
    </OCL.Invariant>
  </OCL.OclModule.ownedElements>
</OCL.OclModule>
```

a) UML object diagram of OCL model transformed from the R2ML model which is shown in Figure 5.18

b) OCL XMI representation of the R2ML model shown in Figure 5.18

Figure 5.21: OCL model and OCL XMI representation of the R2ML model shown in Figure 5.18

The R2ML `AlethicIntegrityRule` element shown in Figure 5.18 is transformed into the OCL `Invariant` element (shown in Figure 5.21), while R2ML `Implication` is transformed into OCL `OperatorCallExp` with the attribute name "*implies*", conforming to the conceptual mappings from Table 4.8. R2ML `AttributionAtom` and `ReferencePropertyAtom` are transformed into the OCL `OperatorCallExp` elements with name "=". R2ML `DatatypePredicateAtom` with `DatatypePredicate` "*greaterThanOrEqual*" is transformed into the OCL `OperatorCallExp` with name ">=". R2ML's `TypedElement` with "*int*" as `Datatype` are transformed into the OCL `IntegerLiteralExp` elements, with certain values. R2ML `Variable` elements are directly transformed into the OCL `Variable` elements that are associated with the `VariableExp` elements through the "*referredVariable*" associations.

The attained OCL model is saved in the model repository and it can be used as an input in other transformations, or it can be serialized into the OCL concrete syntax (EBNF technological space). The serialization can be done by using the defined textual concrete syntax (TCS) and EBNF extractor or automatized AM3 Ant task shown in Figure 5.22.

```
<project name="project" default="extraction">

      <!-- Properties definition -->
      <property name="project.name" value="/TCS/"/>
      <property name="metamodel.name" value="OCL"/>
      <property name="model.name" value="outputOCLModel"/>
      <property name="common.dir" value="common/"/>
      <property name="common.path" value="${project.name}${common.dir}"/>
      <property name="metamodels.path" value="${common.path}metamodels/"/>

      <property name="metamodel.dir" value="${metamodel.name}/"/>
      <property name="metamodel.path" value="${project.name}${metamodel.dir}"/>
      <property name="ebnfinjector.name" value="ebnfinjector"/>
      <property name="ebnfinjector.dir" value="${ebnfinjector.name}/"/>
      <property name="metamodel.jar" value="${metamodel.name}-${ebnfinjector.name}.jar"/>
      <property name="metamodel.uri" value="${metamodel.path}${metamodel.name}.xmi" />
      <property name="model.uri" value="${metamodel.path}${model.name}.xmi" />
      <property name="targetmodel.uri" value="${metamodel.path}${model.name}.ocl" />
      <property name="ebnfnjector.uri" value="${metamodel.jar}" />
      <property name="classname.prefix" value="${metamodel.name}" />
      <property name="metamodel.path" value="${project.name}${metamodel.dir}"/>

      <!-- Targets definition -->
      <target name="extraction">

            <am3.loadModel modelHandler="MDR" name="${metamodel.name}" metamodel="MOF" path="${metamodel.uri}" />
            <am3.loadModel modelHandler="MDR" name="${model.name}" metamodel="${metamodel.name}" path="${model.uri}" />
            <am3.loadModel modelHandler="MDR" name="TCS" metamodel="MOF" path="${metamodels.path}TCS.xmi" />
            <am3.loadModel modelHandler="MDR" name="${metamodel.name}.tcs" metamodel="TCS"
                                              path="${metamodel.path}${metamodel.name}.tcs">
                              <injector name="ebnf">
                                    <param name="name" value="TCS"/>
                              </injector>
            </am3.loadModel>

            <am3.saveModel model="${model.name}" path="${targetmodel.uri}">
                  <extractor name="ebnf">
                        <param name="format" value="${metamodel.name}.tcs"/>
                        <param name="identEsc" value=""/>
                        <param name="indentString" value="  "/>
                        <param name="standardSeparator" value=" "/>
                        <param name="stringDelim" value='"'/>
                  </extractor>
            </am3.saveModel>
      </target>
</project>
```

Figure 5.22: AM3 Ant task which executes the transformation of the OCL model into the OCL code

AM3 Ant task (buildextraction.xml) that is shown in Figure 5.22 can be easily implemented for any meta-model and its textual concrete syntax (TCS). Using EBNF injector, this Ant task loads (`am3.loadModel`) the OCL meta-model, the OCL model for serialization, the TCS meta-model, as well as the defined OCL textual concrete syntax (TCS) for the model (using TCS meta-model). When models and meta-models are loaded, the serialization (`am3.saveModel`) of the given OCL model into the OCL code (EBNF extraction shown in Figure 5.5) is done by using EBNF extractor and OCL textual concrete syntax (TCS).

The result of the OCL model serialization with the AM3 Ant task is the OCL invariant in the EBNF concrete syntax (see Figure 5.23). This process of serializing an OCL invariant from its abstract syntax into the OCL concrete syntax is described in section 4.5.1 and shown in Figure 4.2 (transformation 9. OCL meta-model into the OCL Text).

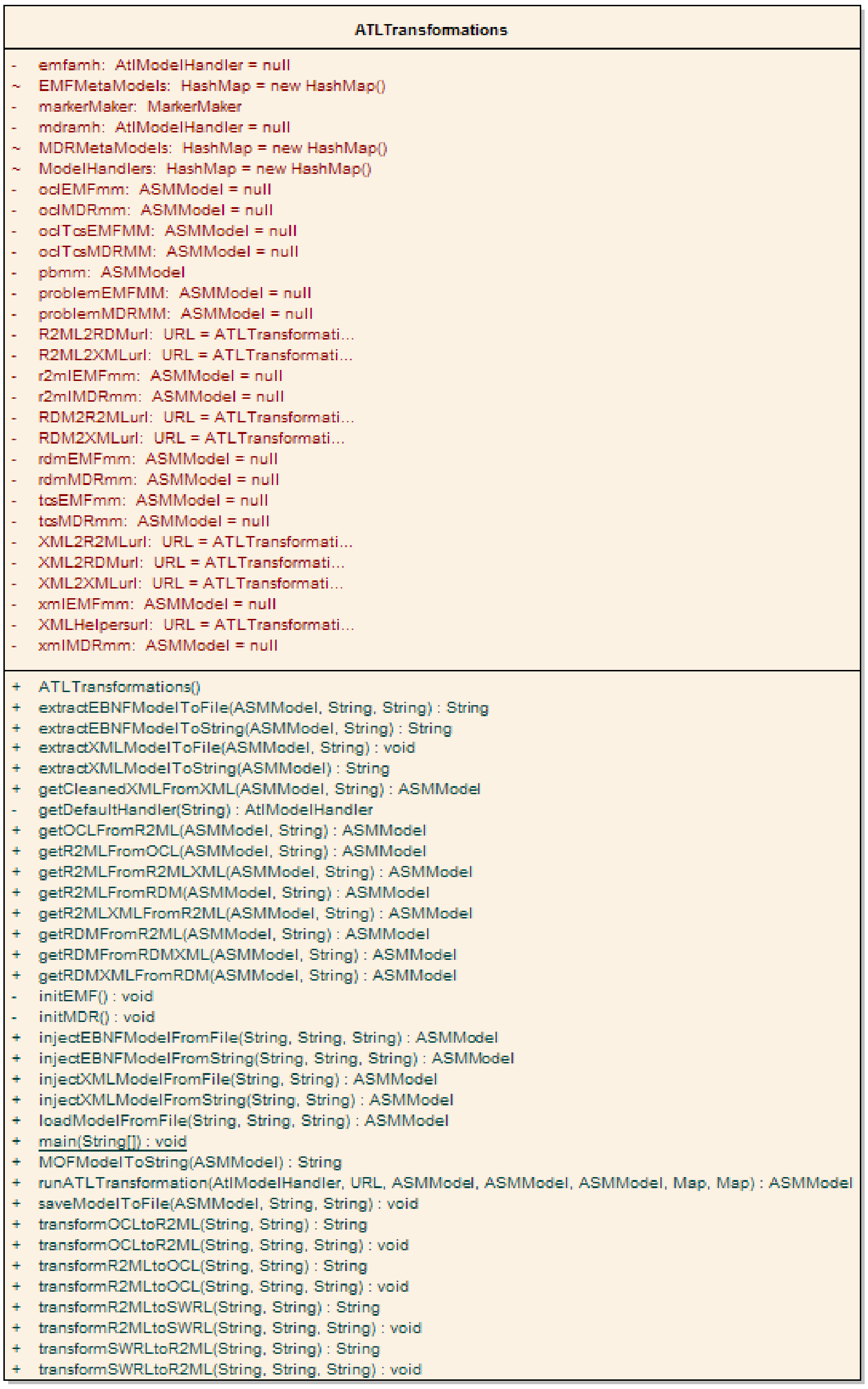
```
context Customer
     inv: self.hasStatus = gold and self.hasTotalPurchase = total and
        total >= 500 implies self.hasDiscount = 10
```

Figure 5.23: OCL invariant serialized from the OCL model shown in Figure 5.19

The OCL invariant shown in Figure 5.23 can be applied on any UML model that contains elements (attributes and references) shown in this invariant.

5.3. IMPLEMENTATION OF TRANSFORMATION EXECUTIONS IN APPLICATION

Building and executing ATL transformations is done within Eclipse integrated development environment. Every time an ATL transformation (i.e., its source code) is saved, a compiled *.ASM* file with the same name is created. Such file is executed in ATL engine. To be able to execute ATL transformations outside the Eclipse environment, i.e. in a desktop or Web-based Java application, as well as through Web services, we have developed the `ATLTransformations` Java class that supports this process (Figure 5.24).



```
ATLTransformations
-  emfamh: AtlModelHandler = null
~  EMFMetaModels: HashMap = new HashMap()
-  markerMaker: MarkerMaker
-  mdramh: AtlModelHandler = null
~  MDRMetaModels: HashMap = new HashMap()
~  ModelHandlers: HashMap = new HashMap()
-  oclEMFmm: ASMModel = null
-  oclMDRmm: ASMModel = null
-  oclTcsEMFMM: ASMModel = null
-  oclTcsMDRMM: ASMModel = null
-  pbmm: ASMModel
-  problemEMFMM: ASMModel = null
-  problemMDRMM: ASMModel = null
-  R2ML2RDMurl: URL = ATLTransformati...
-  R2ML2XMLurl: URL = ATLTransformati...
-  r2mlEMFmm: ASMModel = null
-  r2mlMDRmm: ASMModel = null
-  RDM2R2MLurl: URL = ATLTransformati...
-  RDM2XMLurl: URL = ATLTransformati...
-  rdmEMFmm: ASMModel = null
-  rdmMDRmm: ASMModel = null
-  tcsEMFmm: ASMModel = null
-  tcsMDRmm: ASMModel = null
-  XML2R2MLurl: URL = ATLTransformati...
-  XML2RDMurl: URL = ATLTransformati...
-  XML2XMLurl: URL = ATLTransformati...
-  xmlEMFmm: ASMModel = null
-  XMLHelpersurl: URL = ATLTransformati...
-  xmlMDRmm: ASMModel = null

+  ATLTransformations()
+  extractEBNFModelToFile(ASMModel, String, String) : String
+  extractEBNFModelToString(ASMModel, String) : String
+  extractXMLModelToFile(ASMModel, String) : void
+  extractXMLModelToString(ASMModel) : String
+  getCleanedXMLFromXML(ASMModel, String) : ASMModel
-  getDefaultHandler(String) : AtlModelHandler
+  getOCLFromR2ML(ASMModel, String) : ASMModel
+  getR2MLFromOCL(ASMModel, String) : ASMModel
+  getR2MLFromR2MLXML(ASMModel, String) : ASMModel
+  getR2MLFromRDM(ASMModel, String) : ASMModel
+  getR2MLXMLFromR2ML(ASMModel, String) : ASMModel
+  getRDMFromR2ML(ASMModel, String) : ASMModel
+  getRDMFromRDMXML(ASMModel, String) : ASMModel
+  getRDMXMLFromRDM(ASMModel, String) : ASMModel
-  initEMF() : void
-  initMDR() : void
+  injectEBNFModelFromFile(String, String, String) : ASMModel
+  injectEBNFModelFromString(String, String, String) : ASMModel
+  injectXMLModelFromFile(String, String) : ASMModel
+  injectXMLModelFromString(String, String) : ASMModel
+  loadModelFromFile(String, String, String) : ASMModel
+  main(String[]) : void
+  MOFModelToString(ASMModel) : String
+  runATLTransformation(AtlModelHandler, URL, ASMModel, ASMModel, ASMModel, Map, Map) : ASMModel
+  saveModelToFile(ASMModel, String, String) : void
+  transformOCLtoR2ML(String, String) : String
+  transformOCLtoR2ML(String, String, String) : void
+  transformR2MLtoOCL(String, String) : String
+  transformR2MLtoOCL(String, String, String) : void
+  transformR2MLtoSWRL(String, String) : String
+  transformR2MLtoSWRL(String, String, String) : void
+  transformSWRLtoR2ML(String, String) : String
+  transformSWRLtoR2ML(String, String, String) : void
```

Figure 5.24: The ATLTransformations Java class for executing ATL transformations

The `ATLTransformations` class, shown in Figure 5.24, implements general concepts for executing ATL transformations. Those general concepts includes: initializing model handlers (`initMDR()`, `initEMF()`), transformations between XML and MOF technological space (`injectXMLModelFromFile()` and `extractXMLModelToFile()`), transformations between EBNF and MOF technological spaces (`injectEBNFModelFromFile()` and `extractEBNFModelToFile()`), and executing ATL transformations (`runATLTransformation(...)`). These methods can be used for any (MOF or ECore-based) meta-model and ATL transformation. Along with implementation of methods for general transformations concepts, we have also implemented specific methods for executing transformations on the RDM, OCL and R2ML models. We have defined methods that transform models in the MOF technological space, through direct execution of the ATL transformations. An example for this type of method is `getR2MLFromRDM()` which executes the RDM2R2ML.atl transformation. We have also defined methods which take SWRL, R2ML and OCL rules in their concrete syntaxes, transform them into instances of appropriate models, then execute transformations between models, and finally serialize them into concrete syntaxes of the target languages. An example for this type of method is `transformSWRLtoR2ML()`, which takes a SWRL rule defined in the XML technological space, and as a result returns an R2ML rule, also in the XML technological space.

Architecture of the concrete Web application that uses the `ATLTransformations` class for executing ATL transformations is shown in Figure 5.25.

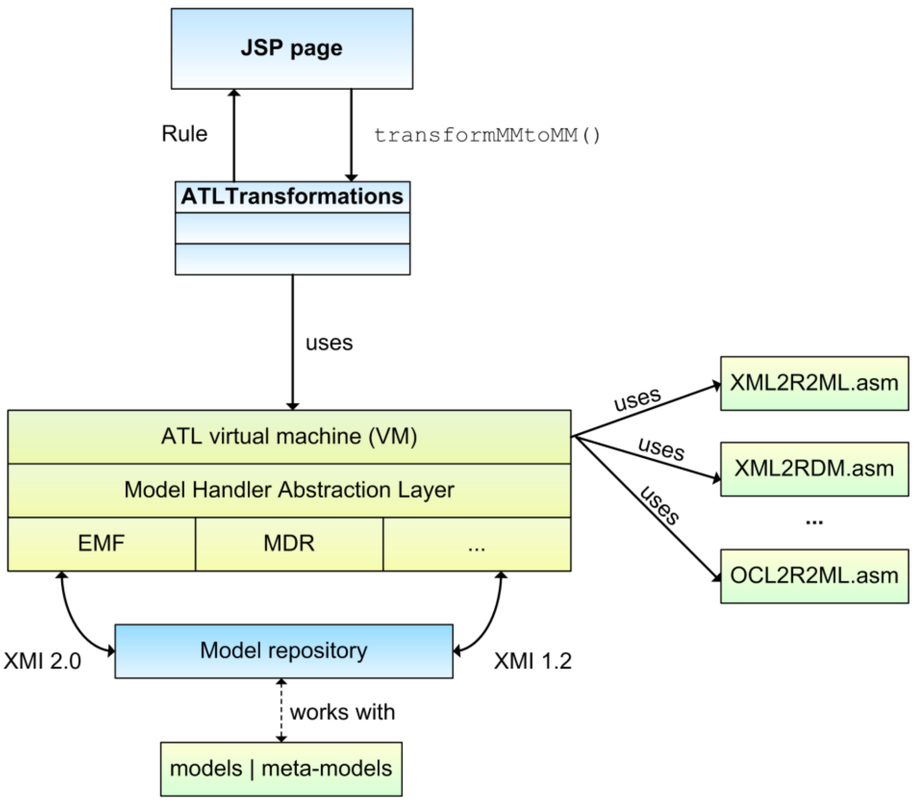


Figure 5.25: Architecture of a Web application that uses `ATLTransformations` class

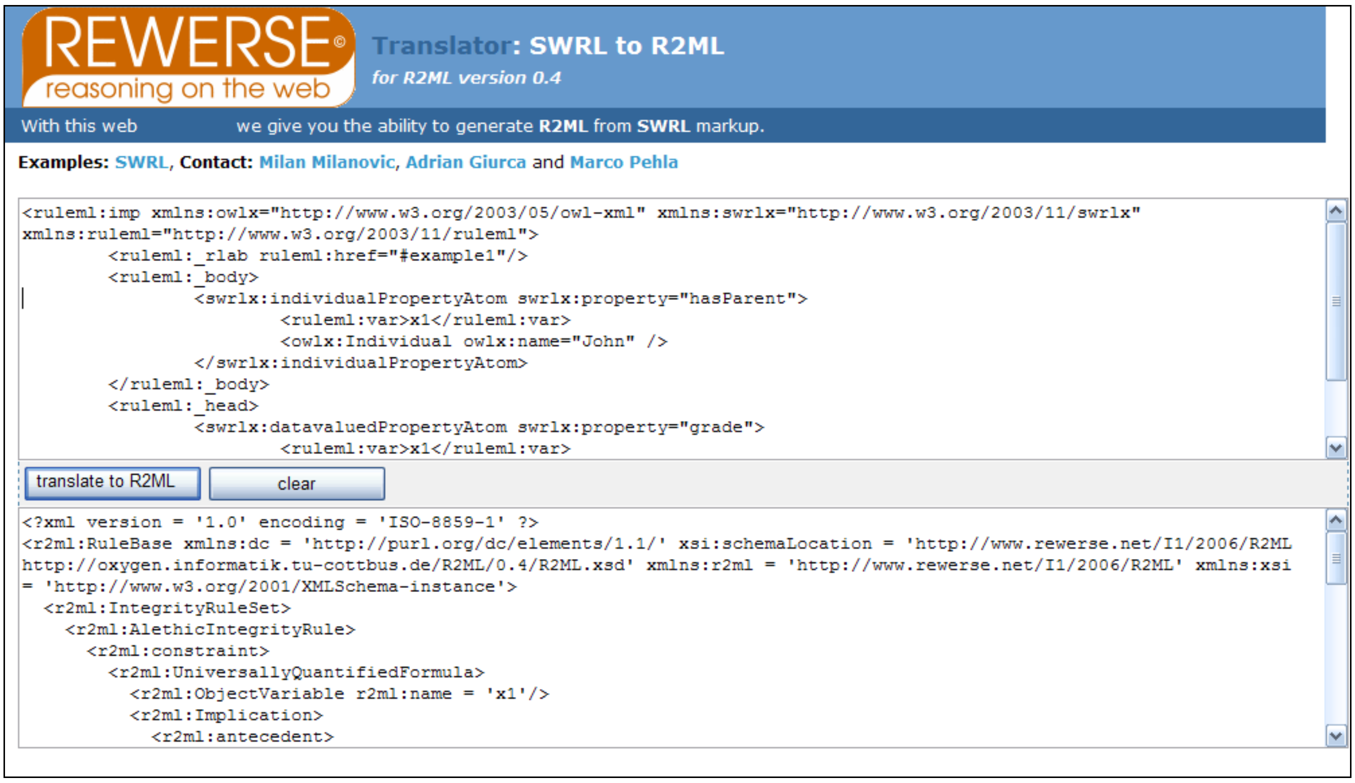
User first enters a rule (SWRL, R2ML or OCL) on the JSP page and runs transformation by clicking the button "Translate to [target language]". JSP page then creates one ATLTransformations object and passes that rule (as string) to the specific method `transformMMtoMM()`. The choosen method invokes methods that execute certain transformations on the input models. For example, the method `transformSWRLtoR2ML()` transforms a SWRL rule into the R2ML rule (see Figure 5.26).

```
/**
 * Transform input SWRL Rule (XML) to output R2ML Rule (XML) - as Strings
 * @param SWRLRuleXML input SWRL XML rule
 * @param modelHandler name of model handler (EMF or MDR)
 * @return output R2ML XML rule
 */
public String transformSWRLtoR2ML(String SWRLRuleXML, String modelHandler) {
      ASMModel xmlModel = injectXMLModelFromString(SWRLRuleXML, modelHandler); // SWRL XML -> XML model
      ASMModel rdmModel = getRDMFromRDMXML(xmlModel, modelHandler);             // XML -> RDM
      ASMModel r2mlModel = getR2MLFromRDM(rdmModel, modelHandler);              // RDM -> R2ML
      ASMModel r2mlXmlModel = getR2MLXMLFromR2ML(r2mlModel, modelHandler);      // R2ML -> XML
      return extractXMLModelToString (r2mlXmlModel);                            // XML -> R2ML XML
} // -- end of transformSWRLtoR2ML
```

Figure 5.26: The `ATLTransformations` method that transforms a SWRL rule into the R2ML rule

These methods return, as their output, the result of executing the transformation, i.e., the resulting rule (string) in its concrete syntax. A JSP page then shows that rule.

By using the `ATLTransformations` class, we have created a concrete Java Web application that executes transformations between SWRL and R2ML rules (and also between OCL and R2ML rules) – R2ML language translator[39] – as a part of the REWERSE project (I1 working group - *Rule Modeling and Markup*[40]). Figure 5.27 shows translator (as JSP page) that transforms a SWRL rule into the R2ML rule[41].



With this web we give you the ability to generate **R2ML** from **SWRL** markup.

Examples: SWRL, **Contact:** Milan Milanovic, Adrian Giurca and Marco Pehla

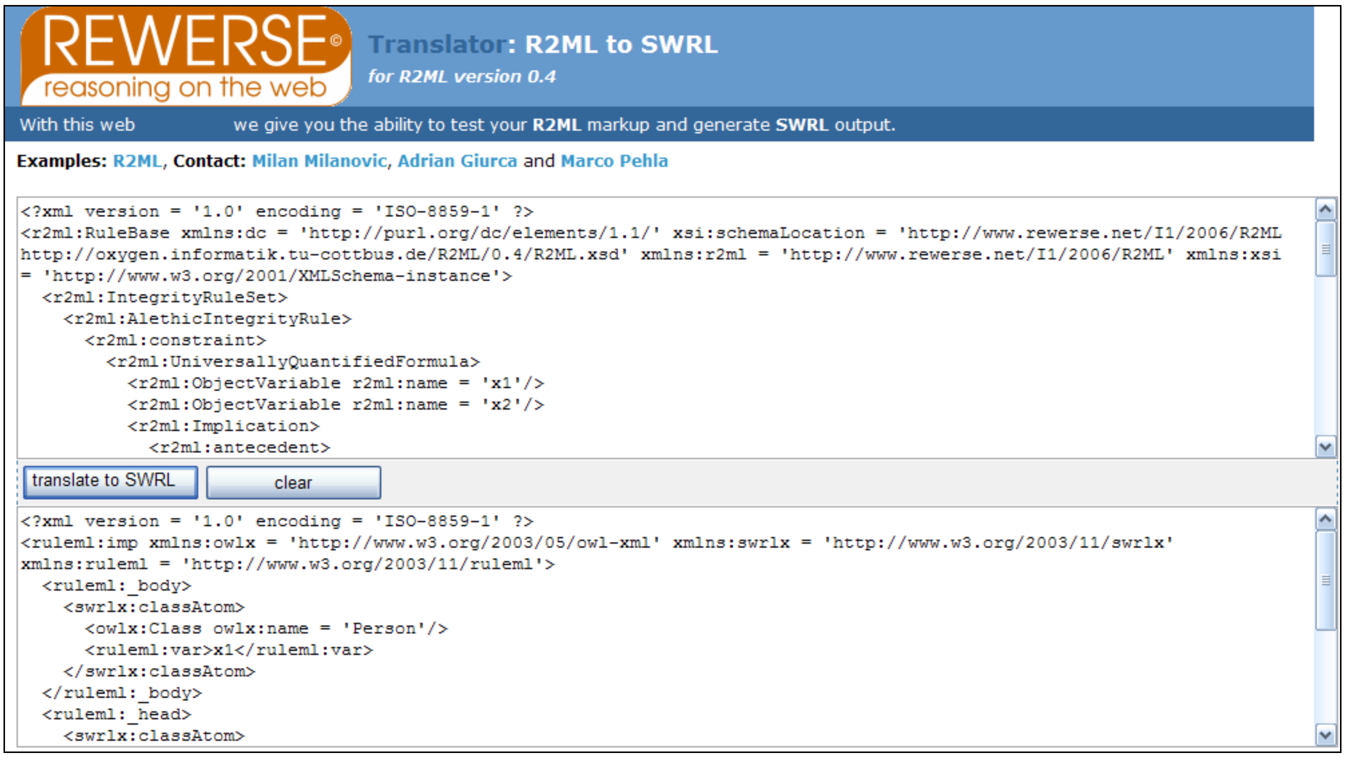
```
<ruleml:imp xmlns:owlx="http://www.w3.org/2003/05/owl-xml" xmlns:swrlx="http://www.w3.org/2003/11/swrlx"
xmlns:ruleml="http://www.w3.org/2003/11/ruleml">
        <ruleml:_rlab ruleml:href="#example1"/>
        <ruleml:_body>
                <swrlx:individualPropertyAtom swrlx:property="hasParent">
                        <ruleml:var>x1</ruleml:var>
                        <owlx:Individual owlx:name="John" />
                </swrlx:individualPropertyAtom>
        </ruleml:_body>
        <ruleml:_head>
                <swrlx:datavaluedPropertyAtom swrlx:property="grade">
                        <ruleml:var>x1</ruleml:var>
```

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<r2ml:RuleBase xmlns:dc = 'http://purl.org/dc/elements/1.1/' xsi:schemaLocation = 'http://www.rewerse.net/I1/2006/R2ML
http://oxygen.informatik.tu-cottbus.de/R2ML/0.4/R2ML.xsd' xmlns:r2ml = 'http://www.rewerse.net/I1/2006/R2ML' xmlns:xsi
= 'http://www.w3.org/2001/XMLSchema-instance'>
  <r2ml:IntegrityRuleSet>
    <r2ml:AlethicIntegrityRule>
      <r2ml:constraint>
        <r2ml:UniversallyQuantifiedFormula>
          <r2ml:ObjectVariable r2ml:name = 'x1'/>
          <r2ml:Implication>
            <r2ml:antecedent>
```

Figure 5.27: Translator - SWRL rule to R2ML rule

[39] http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/15.
[40] http://rewerse.net/I1.
[41] http://oxygen.informatik.tu-cottbus.de/translator/SWRLtoR2ML/index.jsp.

Figure 5.28 shows translator that transforms input a R2ML rule into the analogous SWRL rule[42].


REWERSE
reasoning on the web
Translator: R2ML to SWRL
for R2ML version 0.4
With this web we give you the ability to test your R2ML markup and generate SWRL output.
Examples: R2ML, Contact: Milan Milanovic, Adrian Giurca and Marco Pehla
translate to SWRL
clear

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<r2ml:RuleBase xmlns:dc = 'http://purl.org/dc/elements/1.1/' xsi:schemaLocation = 'http://www.rewerse.net/I1/2006/R2ML
http://oxygen.informatik.tu-cottbus.de/R2ML/0.4/R2ML.xsd' xmlns:r2ml = 'http://www.rewerse.net/I1/2006/R2ML' xmlns:xsi
= 'http://www.w3.org/2001/XMLSchema-instance'>
  <r2ml:IntegrityRuleSet>
    <r2ml:AlethicIntegrityRule>
      <r2ml:constraint>
        <r2ml:UniversallyQuantifiedFormula>
          <r2ml:ObjectVariable r2ml:name = 'x1'/>
          <r2ml:ObjectVariable r2ml:name = 'x2'/>
          <r2ml:Implication>
            <r2ml:antecedent>
```

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<ruleml:imp xmlns:owlx = 'http://www.w3.org/2003/05/owl-xml' xmlns:swrlx = 'http://www.w3.org/2003/11/swrlx'
xmlns:ruleml = 'http://www.w3.org/2003/11/ruleml'>
  <ruleml:_body>
    <swrlx:classAtom>
      <owlx:Class owlx:name = 'Person'/>
      <ruleml:var>x1</ruleml:var>
    </swrlx:classAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:classAtom>
```

Figure 5.28: Translator - R2ML rule to SWRL rule

All of transformations presented in this chapter (i.e., source code) are available for download from ATL transformations web site[43].

[42] http://oxygen.informatik.tu-cottbus.de/translator/R2MLtoSWRL/index.jsp.
[43] http://www.eclipse.org/m2m/atl/atlTransformations.

6. ANALYSIS OF THE PROPOSED SOLUTION

In this chapter, we analyze the proposed rule and knowledge integration. We describe the experiences in the R2ML meta-model development and the transformation use, with a specific attention on the problems that we faced during the development. Besides, we give an overview of the results of metamodel-based transformations of rule languages with suggestions for the future improvements and development. Finally, we compare the implemented solutions with the other existing solutions to the rule interchange.

6.1. DEVELOPMENT AND USE EXPERIENCES

We tested the transformations on a set of real world rules[44] collected by the REWERSE Working Group I1 at the Brandenburg University of Technology at Cottbus. In this section, we report on some lessons that we learned in developing and applying the transformations. These lessons also helped us to validate the R2ML MOF-based meta-model as well as to propose some changes of the R2ML meta-model.

Full MOF compatibility. In order to use the R2ML meta-model with ATL, we had to create it to be fully compliant to the MOF specification. The R2ML meta-model is created in Poseidon for UML[45]. In Poseidon, we first developed the R2ML meta-model as a regular UML model (by using UML elements related to UML class diagrams) that is located on the M1 level of the MDA. Then, we exported the R2ML meta-model into the UML XMI format. To get the R2ML meta-model on the M2 level, we had to transform the UML XMI representation of the metamodel into the MOF XMI representation. To do so, we used the *uml2mof*[46] tool which is distributed with the NetBeans Metadata Repository (MDR). However, in first attempts, we could not import the R2ML meta-model into the model repository, since the meta-model did fully not follow the MOF 1.4 syntax [MOF, 2002]. We performed some changes such as adding names to all unnamed associations and datatype conversions (since MOF does not support all R2ML datatypes). For example, `URIRef` and `UnicodeString` are changed into `String`; and `xs:boolean` is changed into `Boolean`. In this way, the R2ML meta-model has become fully compliant to MOF, so that the R2ML meta-model can be instantiated in model repositories and check whether R2ML models are conformant to the R2ML meta-model. The conversion problem of the UML XMI format into the MOF XMI format can be solved by using MOF-based editor, as MOFLON[47]. MOFLON enables a visual notation for MOF-based meta-models and their export into the MOF 2.0 XMI 2.1 format and the UML 2.0 and XMI 2.1 formats. However, the current MOF 2.0 repository implementations that can use such models are still in phase of early development and they still are not supported by model transformation tools like ATL (thus, we would have to use MOF 1.4 and its MDR model repository). As ATL can use EMF as a model handler, we have been developing R2ML in the MagicDraw tool[48] from R2ML version 0.4. We can do this, since MagicDraw has features for exporting into the EMF UML2 XMI format, and then such exported UML models can be converted into ECore meta-models by using the UML2 plug-in for Eclipse[49]. In this way, we overcame MOF 1.4 constraints (like naming associations, etc.) and the lack of a reliable MOF 2.0 repository implementation [MOF, 2005]. Current MOF 2.0 repository implementations do not have all necessary characteristics for work with MOF 2.0 based models. Examples of such repositories are AIR MDS [Djurić, 2006], which does not have features for exporting models in the MOF XMI format nor

[44] http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/17.
[45] Poseidon for UML, http://www.gentleware.com/.
[46] NetBeans UML2MOF tool, http://mdr.netbeans.org/uml2mof/.
[47] MOFLON, http://www.moflon.org/.
[48] MagicDraw UML, http://www.magicdraw.com.
[49] UML2 Eclipse plug-in, http://www.eclipse.org/modeling/mdt/?project=uml2-uml.

features for code generation; and the MOF2 for Java repository[50], which does not have an event mechanism to allow clients to listen to model changes. Both of these repositories are in the development stages and do not have any industrial or wider community support.

Missing associations. In the implementation of the first transformation, our goal was to transform rules from the R2ML XML format into the R2ML meta-model. This helped us identify some associations missing in the R2ML meta-model without which we could not represent all relations existing in the R2ML XML format. For example, the `IntegrityRuleSet` and `DerivationRuleSet` complex types are sequences of `IntegrityRule` and `DerivationRule` in the R2ML XML schema, respectively. This implicated that in the R2ML meta-model it is necessary to add an association between `IntegrityRuleSet` and `IntegrityRule` as well as another association between `DerivationRuleSet` and `DerivationRule`. Since the R2ML XML format is used for collecting real world rules, and due to the lack of tools that are completely based on R2ML meta-model, this transformation prove a good way for validating R2ML concrete syntax (XML schema) w.r.t. its abstract syntax (i.e., meta-model).

Abstract classes. Originally (in version 0.2), some classes of the R2ML meta-model were defined as abstract classes (e.g., `Disjunction`, `Conjunction`, and `Implication`) [Wagner et al., 2005]. When we attempted to transform rules form the R2ML XML format into the R2ML meta-model, we faced the problem that ATL engine refused executing the ATL transformation. The problem was that some classes should not actually be abstract, as the NetBeans Metadata Reporitory (MDR) model repository prevented their instantiation by strictly following the R2ML meta-model definition. This was an obvious indicator to change such classes not to be abstract.

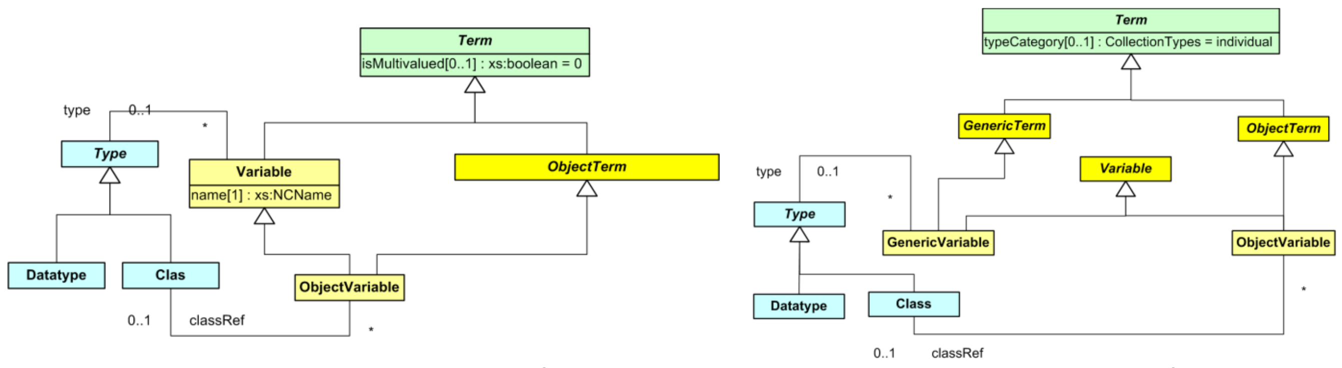
Conflicting compositions. Since the meaning of MOF compositions is fully related to instances of classes connected by compositions, it is very hard to validate the use of compositions in MOF-based meta-models without instantiating meta-models. This means that for a class A that composes a class B, an instance of the class B can be only composed by one and only one instance of the class A. It is also correct to say that a class C also composes the class B. However, the same instance of the class B cannot be composed by two other instances, regardless of the fact that one of them is a instance of the class A and another one of the class C [Bock, 2004]. Since ATL uses the NetBeans Metadata Repository (MDR) for storing input and output models of transformations, MDR does not allow us to execute ATL transformations that break the MOF semantics including the part related to compositions. This actually helped us identify some classes in the R2ML meta-model breaking this rule. To overcome this problem, we have changed (“relaxed”) the composition with a regular association relation. This makes sense, since a variable should be declared once, while all other elements should refer to that variable (not compose it).

Unique class names. In MOF, it is valid to define several classes with the same name provided that they are defined in different packages, i.e., namespaces. In the R2ML meta-model, the `Conjunction`, `Disjunction`, `Negation`, `NegationAsFailure`, and `StrongNegation` classes were defined in two packages, namely, the *formulas* and *formulas.qf* packages. However, the ATL engine (in its current implementation) takes all packages in a meta-model as a unique namespace. This means, there is no way to explicitly select the class by using the namespace it belongs to. In the transformations developed, the ATL engine always used only classes from the *formulas* package. We have overcome this issue by prefixing class names of the *formulas.qf* package with *QF* (e.g., `QFConjunction`). We do not find this change as a problem in the R2ML meta-model, but rather as a benefit that improves the clarity of constructs defined in the meta-model.

Multiple inheritance conflict. During the implementation of the injection and transformation from the R2ML XML to the R2ML meta-model, we noticed the well-known "diamond" problem [Simons, 2005], i.e., a multiple inheritance conflict, in the object-oriented paradigm. Such a conflict arises when a class, say N, obtains the same attribute *attr* from two or more parent class; let us say

[50] http://www2.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava/tool.html.

classes A and B. These both parent classes A and B have the same parent class C from which both of them inherit *attr*, thus there is a conflict to determine from which of them the attribute is inherited and how to access it in the class N. In the previous version of the R2ML meta-model (before 0.4), we have defined three types of `Variable`-s: `ObjectVariable`, `DataVariable`, and `Variable` where `Variable` was the parent of both `ObjectVariable` and `DataVariable`. A problem occurred because `ObjectVariable` inherited both `ObjectTerm` (which inherited `Term`) and `Variable` (which also inherited `Term`), as it is shown in Figure 6.1a. In this way, `ObjectVariable` inherited the class `Term`'s attributes (i.e., *isMultivalued*) from two parents, namely, `ObjectTerm` and `Variable`. The same situation was with `DataVariable` and `DataTerm`. We solved this situation (Figure 6.1b), as follows. First, we introduced the `GenericTerm` class, which inherits the `Term` class, and the `GenericVariable` class that inherits `GenericTerm`. Next, we changed the `Variable` class, which is now an abstract class and it is a parent class for the `GenericVariable` and `ObjectVariable` classes. The `Variable` class does not intert `Term` anymore. In this way, `ObjectVariable` only inherits `Term`'s attributes from one parent only (`ObjectTerm`). Finally, we should note that we have a similar solution for `DataVariable`.



a) The multiple inheritance conflict b) The multiple inheritance conflict solution

Figure 6.1: The multiple inheritance conflict for R2ML `Variable` element (a) and its solution (b)

6.2. META-MODEL TRANSFORMATION RESULTS

The goal of the proposed solution in this thesis is to enable rule interchange between different rule languages via one general rule markup language (R2ML), by using MDE techniques. The main advantage of this approach is that conceptual mappings between languages are specified on the level of their abstract syntax (i.e., meta-models). In this way, it allows us to focus on language concepts, rather than on the implementation details of these languages concrete syntax, which stems from the use of the MDE principles.

In this thesis, we have demonstrated potentials of model transformations for transforming rule languages. First, the use of model transformation languages forces us to use valid source and target models. This means that the transformation cannot be executed properly if either of rule models is not fully conformant to its meta-model (i.e., R2ML models must conform to the R2ML meta-model). Second, every time when a model transformation is executed, the elements of the target model are instantiated in the model repository, and that provided us with the mechanism for instantiation for meta-model. This helped us detect some issues in the R2ML meta-model such as conflicting compositions and inappropriate abstract classes in the R2ML meta-model. Third, instances of rule language meta-models are stored into MOF-based repositories such as MDR. Since model repositories have generic tools for exporting/importing models and meta-models in the XMI format, it is possible to share models with other MOF-compliant applications.

The transformations between the models in R2ML and SWRL (i.e., meta-models) abstract syntax fully captures the definition of SWRL, so that all SWRL constructs can be translated onto their counterparts in R2ML and they can then also be transformed back from R2ML to SWRL. However, we have not yet finalized the implementation of all OWL (i.e., ODM) constructs to their R2ML equivalents (basically this is just for classes and properties – elements for defining vocabularies, while restrictions have already been covered). Having in mind the nature of *open-world inference* that OWL is based on, this is allowed in both SWRL and R2ML. Nevertheless, this may have consequences if we want to map such SWRL rules from R2ML, for example, into OCL constraints for which we strictly have to define all elements of the underlying vocabulary (because OCL and UML are based on *close-world inference* principles). SWRL rule from Figure 4.12 do not have information about the first *individualPropertyAtom*'s property (i.e., *hasParent*), that is, there is no explicitly defined domain and range for this property. An R2ML rule which is obtained from such a SWRL rule cannot be translated into a valid OCL, since we cannot determine the context of the OCL invariant. To solve this problem, we need to define all the properties referred to in the original SWRL rules and transform them to elements of the R2ML `Vocabulary`, as it is done for the SWRL rule from Figure 5.14. On the other hand, the transformation of OCL rules to R2ML, and then in SWRL is possible without similar constraints.

The transformation implementations transform all SWRL and OCL rules to R2ML integrity rules. However, some SWRL rules may be supposed to represent explicit definitions of concepts, so they should be transformed into R2ML derivation rules. The same is true for derivation rules in OCL (*derive*). While a derivation rule represents an explicit constructive definition, an integrity rule rather complements a definition by defining the admissible knowledge states with respect to the concepts constrained by it. While the conditions of a derivation rule are instances of the `AndOrNafNegFormula` class, representing quantifier-free logical formulas with conjunction, disjunction, and negation; conclusions are restricted to quantifier-free disjunctive normal forms without *NAF* (Negation as Failure) [Wagner, 2003]. Generally, supporting the transformation of SWRL rules into R2ML derivation rules will only require using a different type of logical formulas, but much of the current transformation will be reused. Once we support derivation rules, it will be possible to translate SWRL rules into F-Logic, Jess, RuleML, since the present R2ML translators support transformation of derivation rules[51]. Nevertheless, there is an open issue how to determine automatically whether we should translate a SWRL rule into an integrity rule or into a derivation rule. This basically requires analyzing the context in which the rule is defined (i.e., based on the notion of ontology elements that the rule is based on).

If we have in mind the changes that we have done in the OMG's OCL meta-model [OCL, 2006], i.e., the adding of a new package *EnhancedOCL* (shown in section 3.3.4.3), which are caused by ANTLR limitations, adding class for representing invariants (e.g. Invariant class), qualify this changed meta-model to be used in modeling and transformations. With new versions of ANTLR and TCS, as well as with new OMG's OCL meta-model specifications, these problems could be exceeded, and the OCL textual concrete syntax can be more precisely defined for mapping to the OCL meta-model, with no major changes of the OCL meta-model.

6.3. COMPARISON OF THE REALIZED WITH EXISTING SOLUTIONS

An important characteristic of Semantic Web rule languages is that they do primarily not deal with the interchange of rules between various types of rules on the Web. This means that Semantic Web rule languages do not tend to compromise their reasoning characteristics for the broader syntactic expressivity. However, there are many Semantic Web applications that may use (OWL) ontologies, but whose business logic is implemented in different rule languages (e.g., F-Logic, Jess and Prolog). In this

[51] http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/15.

case, the main goal is an interchange rule language. This is actually the main focus on the of research on the Web and rule languages that is chiefly articulated through the W3C effort called Rule Interchange Format (RIF) [Ginsberg, 2006], while the most known efforts in that area are the RuleML language [Boley et al., 2001] and R2ML [Wagner et al., 2005]. The primary result expected from this research stream is to define an XML-based concrete syntax for sharing rules on the Web.

To the best of our knowledge, there is no any solution to transforming rule languages based on model transformation languages. Most of previous solutions to transforming rule languages such as RuleML and SWRL are implemented by using XSLT or programming languages (Java) [Grosof et al., 2002] [Ball et al., 2005] [Gandhe et al., 2002]. By the nature, our solution is most similar to those based on the use of XSLT, as a general purpose transformation language for the XML technological space. Examples of transformations for the RuleML language, which are done by using XSLT approach, are the following: XSLT translator[52] between the Horn-logic subsets of RuleML and Relational-Functional Markup Language (RFML) [RFML, 2006], XSLT translator from RuleML to Jess[53], translators between Positional RuleML to Object-Oriented RuleML[54], and translators between OWL and RuleML[55] and SWRL and RuleML[56]. Name also translators for R2ML that are developed by using XSLT[57] such as translators from R2ML to F-Logic, between the F-Logic XML format and R2ML, from R2ML to Jess (rule engine), from R2ML to RuleML, and from R2ML to rules of Jena2 – a popular Semantic Web framework [McBride, 2002].

Although the use of XML is very suitable, the previous analysis of the use of XSLT for sharing knowledge indicates that XSLT is hard to maintain where modifications of input and output formats can completely invalidate previous versions of XSLTs [Jovanović & Gašević, 2005]. Even some recent experiences in transforming rule languages (SWRLp) report on constraints of XSLT (e.g., to transform unique symbols) that can only be overcome by XSLT extensions implemented in other languages such as Java and Jess [Matheus, 2004]. In our case, we used the ATL language whose main benefits are a good support of different technical spaces through the XML injection and extraction and advanced features for creating and using a richer set of transformation rules (i.e., matched, called, lazy, and unique). For example, one can define transformations that specify their own sequence of the rule executions (by using ATL entry-point and endpoint constructs), so that one does not have to follow the execution of rules based on an internal resolution algorithm of ATL (e.g., the one that is being used for matched rules in ATL).

The OMG's Ontology Definition Metamodel (ODM) specification [ODM, 2006] [Gašević et al., 2006] is closely related to the presented work in this thesis, as it specifies MOF-based meta-models for Semantic Web ontology languages Resource Description Framework Schema (RDFS) and OWL, i.e., it defines the abstract syntax of RDFS and ODM by using MOF. The ODM specification also defines QVT-based mappings between the ODM and RDFS meta-models with meta-models of languages such as UML, Common Logics, Topic Maps, and Entity-Relationship models. However, all these transformations are defined on the level of models, thus everything happens in the MOF technical space [Gašević et al., 2005]. This means that, for example, there is a gap between the RDF/XML syntax [Beckett, 2004] usually supported by OWL tools (as a concrete syntax of the OWL language) and the OWL meta-model (i.e., an abstract syntax of OWL). The approach proposed in this thesis shows how this can be overcome, so that one can achieve the full compatibility between abstract and concrete syntax of a rule languages for knowledge representation.

[52] Translators between RuleML and RFML, http://www.relfun.org/ruleml/rfml-ruleml.html.
[53] Translator RuleML to Jess, http://www.ruleml.org/jess.
[54] Object-oriented into positional RuleML translator, http://www.ruleml.org/ooruleml-xslt/oo2prml.html.
[55] Transformational implementations of the OWL Semantics, http://www.ag-nbi.de/research/owltrans.
[56] An Engine for SWRL rules in RDF graphs, http://www.ag-nbi.de/research/swrlengine.
[57] The REWERSE I1 Rule Markup Language (R2ML) Translators, http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6.

The current status of transformations for certain rule types between the R2ML language and other rule languages is shown in Table 6.1. Transformations between R2ML and RuleML, Jess, F-Logic, F-Logic XML, and Jena are implemented just in one direction (symbol ⇒ in the table), while transformations between R2ML and SWRL and OCL are implemented in both directions (symbol ⇔ in the table). For R2ML, transformations between the R2ML XML-based concrete syntax and the R2ML abstract syntax (i.e., meta-model) are realized in both directions. Note that all the QVT/ATL transformations from Table 6.1 are defined in this thesis and they are in detail described in Chapter 4.

Table 6.1: Transformations of between R2ML and other rule languages

R2ML	**RuleML**	**Jess**	**F-Logic**	**F-Logic XML**	**Jena**	**SWRL**	**OCL**
Derivation	⇒	⇒	⇒	⇒	⇒		
Integrity						⇔	⇔
Reaction							
Production	⇒	⇒			⇒		
Transformation Language	*XSLT*					*QVT/ATL*	

From Table 6.1, we can see that for the complete integration between different rule languages and R2ML, we need to support the same rule types, e.g., support for derivation rule in SWRL and OCL. As two most known proposals for RIF, R2ML and RuleML support different rule types, and for both languages there are already implemented transformations with other rule languages. Table 6.2 shows languages (in columns) for which transformations are implemented with RuleML and R2ML (support for SWRL implies OWL, on which SWRL is built). Labels ⇒, ⇔ and ⇐ are directions of realized transformations if we start from languages shown in rows (R2ML and RuleML), while with characters in brackets below arrows is denoted which rule types these transformations support. For RuleML, translators between different versions of this language are implemented (e.g., translator of RuleML 0.88 to RuleML 0.91). RuleML is based on Datalog, and it currently supports integrity rules, derivation rules, reaction rules, production rules, and transformation rules.

Table 6.2: Transformations of the R2ML and RuleML with other rule languages (rule types supported by these transformations are: *I* - integrity rules, *D* - derivation rules, *R* - reaction rules, *P* - production rules and *T* - transformation rules)

	R2ML	**RuleML**	**Jess**	**F-Logic**	**Jena**	**SWRL**	**OCL**	**POSL**	**RFML**
R2ML		⇒ (*D*, *P*)	⇒ (*D*, *P*)	⇒ (*D*)	⇒ (*D*, *P*)	⇔ (*I*)	⇔ (*I*)		
RuleML	⇐ (*D*, *P*)		⇒ (*P*, *R*)			⇐ (*D*)		⇔ (*D*)	⇔ (*D*)

From Table 6.2, we can see that for R2ML and RuleML there are transformation implemented for Jess and SWRL, while for other languages transformations are implemented only for one of these two languages. However, as we have transformations between R2ML and RuleML, it is possible to integrate all these languages with the R2ML language, but one limitation is to use the same rule types (for example, to share rules between SWRL and F-Logic via R2ML, both of these languages should have defined transformations for integrity or derivation rules with R2ML). From the table above, we can also see that transformations for R2ML integrity rules are defined only for SWRL and OCL languages, thus this thesis is the first one that covers the problem of transformations of integrity rules.

7. CONCLUSIONS

In this chapter, we show results that are achieved by the research presented in this thesis. We also comment on the potentials of practical use of developed meta-model and transformations and give some reflections on the future plans.

7.1. ACHIEVED CONTRIBUTIONS

The research and realized solution that is shown in this thesis had two objectives: first, development and second, presentation of a general rule markup language for representing rules on the Semantic Web by using the Model Driven Engineering (MDE) approach. This resulted in the development of the rule language's abstract and concrete syntax through collaborative work with REWERSE I1[58] and LORNET Theme 1[59] projects, as well as the development of a software environment for rule interchange between this general rule language and other rule languages. Contributions of this research are given in the rest of the section.

In this thesis, we can emphasize the following conceptual contributions:

- Overview and analysis of disciplines which are relevant for the subject of the research, namely: the concept of ontologies is described, the basic concepts of the Semantic Web and Model Driven Engineering are defined, and an overview of basic rule languages on the Web is also given;
- Overview of the design of a number of software and conceptual environments and modeling tools;
- Contributions to the general rule markup language for representing rules on the Web (concrete and abstract syntax), named REWERSE I1 Rule Markup Language (R2ML);
- Overview of experiences in developing and designing the R2ML language (i.e., meta-model);
- Extension of the basic concepts of the OCL and RDM meta-models;
- Conceptual solutions for mappings between R2ML integrity rules and SWRL rules and between R2ML integrity rules and OCL invariants;
- Design and analysis of the bridging different technological spaces in the MDA environment, on the problem of R2ML-based rule interchange;
- Design of an abstract software environment for the use of QVT (ATL) based transformations in Java applications;
- Comparative analysis of the R2ML and RuleML languages, as well as defining mappings between the two lanuages;
- Comparative analysis of the proposed approach to sharing rules between different rule languages with other existing solutions.

Practical contributions of this thesis are following:

- Analysis of existing technologies that can be used in developing a general rule markup language (R2ML);
- Comparative analysis of existing model transformations and modeling tools;
- Design and development of an infrastructure for development of the R2ML meta-model and R2ML XML Schema;
- Adopting the R2ML meta-model to model handlers and repositories;
- Implementation of the RDM (with ODM), OCL, XML and R2ML meta-models;

[58] http://rewerse.net/I1/.
[59] http://lornet.iat.sfu.ca/.

- Implementation of bidirectional transformations between the models in: the R2ML XML-based concrete syntax and the R2ML meta-model, the SWRL (OWL) XML-based concrete syntax and the RDM meta-model, the RDM meta-model and the R2ML meta-model, the OCL meta-model and the R2ML meta-model, and the OCL meta-model and the OCL textual concrete syntax;
- Definition of the OCL textual concrete syntax, and consequently, generation of the ANTLR-based OCL grammar, OCL parser and OCL lexer;
- Implementation of a Java-based Web application for translating rules;
- Analysis and detail testing of recommended solution;
- Recommendation for further work directions based on presented research.

7.2. USAGE DOMAIN

The general rule markup language R2ML has several domains of use: it can be used as a central rule language between different systems and tools; it can be also used for expressing derivation rules over Web ontologies by adding definitions of derivation concepts or for defining data access rules; it can be used for describing and publishing reaction properties of systems in a form of reaction rules; it can be used for modeling the Semantic Web services and it can give complete XML-based specification of a software agent.

An important advantage of the solution presented is a definition of the R2ML abstract syntax, which is defined as a meta-model by using the Meta-Object Facility (MOF) modeling language. In this way, we enabled rule definition and work with R2ML by using standard MOF and UML developing tools, as well as state-of-art model repositories (handlers).

The R2ML language along with the mappings and transformations realized to other rule languages, are primary designed for defining different rule types and their interchange. Thus, this helped us represent rules defined in different rule languages in a unique way. By using the MDE principles, it is possible to integrate rules that are defined in languages whose concrete syntax is defined in different technological spaces in a unique way. We should notice that the R2ML language is not limited to transformations between represented languages only, and that its development continues (actual version is 0.5 beta).

Using and sharing rule on the Semantic Web, as one of the main integration and standardization challenges, as well as reasoning on ontologies, is enabled in simple way by using the solution that is represented in this thesis. Use of this solution is possible anywhere where knowledge integration is necessary, and specially in intelligent information system, in (Semantic) Web services development and on the Semantic Web.

7.3. FURTHER WORK AND RESEARCH DIRECTIONS

The further research and development plans may comprise:

- Improvement and update of the actual version of the R2ML language:
    - adding a new symbol as a non-abstract negation concept that will be a super-concept of the StrongNegation and NegationAsFailure concepts;
    - extending the RuleBase element to support knowledge bases with references to external knowledge bases;
    - allowing derivation rules definitions to be partial or complete;
    - allowing atoms to have a certain level of uncertainty as special case of generic qualification;
    - creating new Event and AtomicEventExpr meta-models;
    - supporting transformation rules in R2ML.

- Support of the SWRL RDF/XML concrete syntax in the XML2RDM and RDM2XML transformations (3. XML2RDM and 4. RDM2XML from Figure 4.2, that are described in sections 4.2.2.1 and 4.2.2.2, respectively), and for the RuleML 0.9 tags;
- Support for automatic generation of domain meta-models based on an arbitrary XML Schema (i.e., XML Schema meta-model);
- Implement transformations between features for defining vocabularies of UML, R2ML and OWL;
- Develop an algorithm that will recognize derivation rules in the SWRL language to be added to the R2ML2RDM transformation (6. R2ML2RDM shown in Figure 4.2, that is described in section 4.3.2);
- Implement a transformation of R2ML integrity rules into the R2ML derivation rules;
- Implement an RuleML meta-model and QVT-based transformation between the RuleML model and the R2ML model, which will enable mapping between these two most important RIF proposals, on the level of their abstract syntaxes (i.e., meta-models);
- Create mappings between R2ML and the OMG’s initiative for business vocabulary semantics and business rules (SBVR) and production rule, which will enable further integration between Semantic Web standards and software engineering standards;
- Enable policy interchange on the Semantic Web by using R2ML. Create transformations between R2ML and policy rule languages as KAoS and Rei, and then enable policy modeling by using OCL and UML with transformations between OCL and R2ML realized in this thesis;
- Create a software environment as an Java application with a GUI, by using AndroMDA, EMF, GEF and GMF as tools for generation of complete applications from domain models, that will enable following:
    - integration of ontologies and rules by using QVT-based transformations;
    - creation of automatic transformations between XML Schemas and meta-models, as well as translation of existing XSLT transformation in set of QVT-based rules;
    - automatic generation of XML-based concrete syntax based on their abstract syntax, and vice-versa;
    - generation of meta-models based on ontologies, and ontologies based on meta-models, and discovery of relations between concepts of ontologies and meta-models;
    - publishing the middle (business) layer of such application as Semantic Web service.

LITERATURE

[Allilaire et al., 2006] Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I., "*ATL - Eclipse Support for Model Transformation*", In: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France, 2006.

[Alanen et al., 2003] Alanen, M., Lilius, J., Porres, I., Truscan, D., "*Realizing a Model Driven Engineering Process*", TUCS Technical report No 565, 2003.

[AM3, 2007] AM3 Official site [Online]. Available: http://www.eclipse.org/gmt/am3.

[Anutariya, 2001] Anutariya, C., "*Semantic Web Modeling and Programming with XDD*". SWWS'01: The First Semantic Web Working Symposium. [Online]. Available: http://www.semanticweb.org/SWWS/program/full/paper17.pdf.

[Atkinson & Kühne, 2003] Atkinson, C., Kühne, T., "*Model-Driven Development: A Metamodeling Foundation*", IEEE Software (spec. issue on Model-Driven Development), Vol. 20, No. 5, pp. 36.-41., September/October 2003.

[ATL, 2006] "*Atlas Tansformation Language - User Manual*", version 0.7, ATLAS group, Nant. [Online]. Available: http://www.eclipse.org/gmt/atl/doc/ATL_User_Manual[v0.7].pdf.

[ATL, 2007] ATL Official site [Online]. Available: http://www.eclipse.org/gmt/atl.

[ATLVM, 2005] Specification of the ATL Virtual Machine [Online]. Available: http://www.eclipse.org/gmt/atl/doc/ATL_VMSpecification[v00.01].pdf.

[Ball et al., 2005] Ball, M., Boley, B., Hirtle, D., Mei, J., Spencer, B., "*Implementing RuleML Using Schemas, Translators, and Bidirectional Interpreters*", In Proc. of the W3C Workshop on Rule Languages for Interoperability, Washington, D.C., 2005.

[Beckett, 2004] Beckett, D., Ed., "*RDF/XML Syntax Specification (Revised)*", W3C Recommendation, http://www.w3.org/TR/rdf-syntax-grammar/.

[Berners-Lee, 1998] Berners-Lee, T., "*Semantic Web Road Map*", W3C Design Issues. [Online]. Available: http://www.w3.org/DesignIssues/Semantic.html.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., Lassila, O., "*The Semantic Web*", Scientific American, Maj. Vol. 284, No. 5, pp. 34.-43., 2001.

[Bézivin, 2001] Bézivin, J., "*From Object Composition to Model Transformation with the MDA*", In Proc. of the 39th Int. Conf. and Exh. on Tech. of OO Lang. and Sys., pp. 350-355, 2001.

[Bézivin et al., 2003] Bézivin, J., Farcet, N., Jezequel, J-M, Langlois, B., Pollet, D., "*Reflective model driven engineering*", In Proceedings of UML2003, USA, 2003.

[Beydeda et al., 2005] Beydeda, S., Book, M., Gruhn, V., (eds.) "*Model-Driven Software Development*", Springer-Verlag, 2005.

[Bickley & Guha, 2003] Brickley, D., Guha, R., V., “*RDF Vocabulary Description Language 1.0: RDF Schema*”, W3C Working Draft. [Online]. Available: http://www.w3.org/TR/2003/WD-rdf-schema-20031010/.

[Bock, 2004] Bock, C., “*UML 2 Composition Model*", Journal of Object Technology, Vol. 3, No. 10, pp. 47-73.

[Boley et al., 2001] Boley, H., Tabes, S., Wagner, G., "*Design rationale of RuleML: a markup language for semantic web rules*", in Proc. Semantic Web Working Symposium (SWWS '01), Stanford University, 2001.

[Brockmans & Haase, 2006] Brockmans, S., Haase, P., “*A Metamodel and UML Profile for Rule-extended OWL DL Ontologies - A Complete Reference*”, Universität Karlsruhe (TH) - Technical Report, 2006.

[Brown et al., 2005] Brown, W., A., Conallen, J., Tropeano, D., "*Introduction: Models, Modeling, and Model-Driven Architecture (MDA)*", Showed in: Beydeda, S., Book, M., Gruhn, V., (eds.) "*Model-Driven Software Development*", Springer-Verlag, pp. 1.-16., 2005.

[Budinsky et al., 2003] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, J., T., "*Eclipse Modeling Framework: A Developer's Guide*", Addison Wesley, 2003.

[Chandrasekaran et al., 1999] Chandrasekaran, B., Josephson, R., J., Benjamins, R., V., “*What Are Ontologies, and Why Do We Need Them?*”, IEEE Intelligent Systems, Vol. 14, No. 1, 1999.

[Czarnecki & Helsen, 2003] Czarnecki, K., Helsen, S., "*Classification of model transformation approaches*", OOPSLA2003, Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003.

[Conallen, 2002] Conallen, J., "*Building Web Applications with UML*", Addison-Wesley, 2nd Edition, 2002.

[Cranefield, 2001] Cranefield, S. "*UML and the Semantic Web*". SWWS’01: The First Semantic Web Working Symposium. [Online]. Available: http://www.semanticweb.org/SWWS/program/full/paper1.pdf.

[Cost et al., 2001] Cost, R., S., et al. "*ITTALKS: A Case Study in the Semantic Web and DAML*". SWWS’01: The First Semantic Web Working Symposium. [Online]. Available: http://www.semanticweb.org/SWWS/program/full/paper41.pdf.

[Damásio et al., 2006] Damásio, V. C., Analyti, A., Antoniou, G., Wagner, G. "*Supporting Open and Closed World Reasoning on the Web*", In Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR06), Budva, Montenegro, 2006.

[Dean & Schreiber, 2004] Dean, M., Schreiber, G. (eds.), "*OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004*", [Online]. Available: http://www.w3.org/TR/2004/REC-owl-ref-20040210/, 2004.

[Decker et al., 2000] Decker, S., Melnik, S., Van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Ederman, M., Horrocks, I., “*The Semantic Web: The Roles of XML and RDF*”, IEEE Internet Computing, Vol. 4, No. 5, 2000.

[Devedžić, 2001] Devedžić, V., “*Ontologies: Borrowing from software patterns*”, ACM intelligence Magazine, Vol. 10, No. 3, pp. 14-24, 2001.

[Dirckze, 2002] Dirckze, R. (spec. lead), "*Java Metadata Interface (JMI) Specification Version 1.0*", [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html, 2002.

[Djurić, 2006] Djurić, D., "*Semantic Web Models*", PhD thesis (in Serbian), Faculty of Organizational sciences, University of Belgrade, 2006.

[Djurić et al., 2006] Djurić, D., Gašević, D., Devedžić, V., “*The Tao of Modeling Spaces*”, in *Journal of Object Technology*, vol. 5. no. 8, Novmeber-December 2006, pp. 125-147, [Online]. Available: http://www.jot.fm/issues/issue_2006_11/articlex.

[EAI2004] "*UML Profile and Interchange Models for Enterprise Application Integration (EAI) Specification*", OMG Formal Specification, March 2004, formal/04-03-26.

[EMF2005] "*Generating an EMF 2.1 model*" tutorial, [Online]. Available: http://www.eclipse.org/emf/docs.php?doc=tutorials/clibmod/clibmod.html, Eclipse project dokumentacija, 2005.

[Euzenat, 2001] Euzenat, J., "*An Infrastructure for Formally Ensuring Interoperability in a Heterogeneous Semantic Web*”. SWWS’01: The First Semantic Web Working Symposium. [Online]. Available: http://www.semanticweb.org/SWWS/program/full/paper16.pdf.

[Favre, 2004] Favre, J., M., "*Towards a Basic Theory to Model: Model Driven Engineering*", Workshop on Software Model Engineering, WISME 2004, Lisabon, Portugal, 11. oktobar, 2004. [Online]. Available: http://www-adele.imag.fr/~jmfavre/papers/
TowardsABasicTheoryToModelModelDrivenEngineering.pdf

[Favre, 2004-1] Favre, J., M., "*Foundations of meta-pyramids: languages and metamodels – Episode II: story of Thotis the Babbon*", [Online]. Available at http://www-adele.imag.fr/~jmfavre/, 2004.

[Falkenberg, et al., 1998] Falkenberg, E., D., Hesse, W., Lindgreen, P., Nilsson, B., E., Han Oei, J., E., Rolland, C., Stamper, R., K., van Assche, F., J., M., Verrijn-Stuart, A., A., Voss, K., "*A framework of information system concepts*", The FRISCO report, 1998.

[Gandhe et al., 2002] Gandhe, M. Finin, T., & Grosof, B., “*SweetJess: Translating DamlRuleML to Jess*”, In Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web at 1st International Semantic Web Conference, the Sardinia, Italy, 2002.

[Gardner et al., 2003] Gardner, T., Griffin, C., Koehler, J., Hauser, R., "*A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard*", 1st International Workshop on Metamodeling for MDA, York, UK, 2003.

[Gašević, 2004] Gašević, D., "*Petri Net Ontology*", PhD thesis (in Serbian), Faculty of Organizational sciences, University of Belgrade, 2004.

[Gašević et al., 2005] Gašević, D., Djurić, D., Devedžić, V., "*Bridging MDA and OWL ontologies*", Journal of Web Engineering, vol. 4, no. 2, pp. 119-134, 2005.

[Gašević et al., 2006] Gašević, D., Đurić, D., Devedžić, V., "*Model Driven Architecture and Ontology Development*", Springer, 2006.

[Ginsberg, 2006] Ginsberg, A., "*RIF Use Cases and Requirements*", W3C Working Draft, http://www.w3.org/TR/rif-ucr/, 2006.

[GMT, 2006] Eclipse GMT Official site [Online]. Available: http://www.eclipse.org/gmt.

[Grosof et al., 2002] Grosof, B. N., Gandhe, M. D., Finin, T. W., "*SweetJess: Translating DAMLRuleML to JESS*", In Proceedings of the 1st International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy, 2002.

[Gruber, 1993] Gruber, T., "*A translation approach to portable ontology specifications*" Knowledge Acquisition, Vol. 5, No. 2, 1993.

[Happel & Seedorf, 2006] Happel, H., J., Seedorf, S., "*Applications of Ontologies in Software Engineering*", Workshop on Sematic Web Enabled Software Engineering" (SWESE) on the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia, November 5-9, 2006.

[Heflin, 2004] Heflin, J., "*OWL Web Ontology Language: Use Cases and Requirements*". [Online]. Available: http://www.w3.org/TR/webont-req/.

[Hirtle et al., 2006] Hirtle, D., Boley, H., Grosof, B., Kifer, M., Sintek, M., Tabet, S., Wagner, G., "*Schema Specification of RuleML 0.91*", http://www.ruleml.org/spec/, 2006.

[Hori et al., 2003] Hori, M., Euzenat, J., Patel-Schneider, F., P., "*OWL Web Ontology Language XML Presentation Syntax*", W3C Note, 2003.

[Horrocks et al., 2004] Horrocks, I., Patel-Schneider, F., P., Boley, H., Tabet, S., Grosof, B., Dean, M., "*SWRL: A Semantic Web Rule Language, Combining OWL and RuleML*", W3C Member Submission, 21 May, 2004.

[Hughes, 1999] Hughes, R.I.G., "*The Ising model, computer simulation, and universal physics*", In M. Morgan and M. Morrison (eds.), Models as mediators, Perspectives on natural and social science, Cambridge University Press, 1999.

[Hunter, 2001] Hunter, D., "*From begining... XML*" (In Serbian), CET, 2001.

[Jovanović & Gašević, 2005] Jovanović, J., Gašević, D., "*XML/XSLT-Based Knowledge Sharing*", Expert Systems with Applications, Vol. 29, No. 3, 2005, pp. 535-553.

[Jovanović, 2005] Jovanović, J., "*Dynamic creation of educational content on the Semantic Web*", master thesis (in Serbian), Faculty of Organizational sciences, University of Belgrade 2005.

[Jouault & Bézivin, 2006] Jouault, F., Bézivin, J., "*KM3: a DSL for Metamodel Specification*", In Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, pp. 171-185, 2006.

[Jouault et al., 2006] Jouault, F., Bézivin, J., Kurtev, I., "*TCS: Textual Concrete Syntax*", In Proceedings of the 2nd AMMA/ATL Workshop ATLAS group (INRIA & LINA), Nantes, France, 2006.

[Kalfoglou, 2001] Kalfoglou, Y., "*Exploring ontologies*", Handbook of Software Engineering and Knowledge Engineering, Vol. I – Fundamentals, World Scientific Publishing Co, Singapore, 2001.

[Kent, 2002] Kent, S., "*Model Driven Engineering*", In Proceedings of IFM2002, LNCS 2335, Springer, 2002.

[Klein et al., 2000] Klein, M., Fensel, D., Van Harmelen, F., Horrocks, I., "*The Relation between Ontologies and Schema-Languages: Translating OIL Specifications to XML Schema*", In Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods, 14th European Conference on Artificial Intelligence, Berlin, Germany, 2000.

[Klein & Bernstein, 2001] Klein, M., Bernstein, A., "*Searching for Services on the Semantic Web Using Process Ontologies*", SWWS'01: The First Semantic Web Working Symposium. [Online]. Available: http://www.semanticweb.org/SWWS/program/full/paper2.pdf.

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

[Klint et al., 2005] Klint, P., Lämmel, R., Verhoef, C., "*Toward an engineering discipline for grammarware*", ACM TOSEM, 2005.

[Klein, 2001] Klein, M., "*XML, RDF, and Relatives*", IEEE Intelligent Systems, Vol. 16, No. 2, 2001.

[Kogut et al., 2002] Kogut, P., Cranefield, S., Hart, L., Dutra, M., Baclawski, K., Kokar, M., Smith, J., "*UML for ontology development*", The Knowledge Engineering Review, Vol. 17, No. 1, pp. 61.-64., 2002.

[Kühne, 2006] Kühne, T., "*Matters of (Meta-)Modeling*", Journal on Software and Systems Modeling, Volume 5, Number 4, 369-385, December 2006.

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., Aksit, M., "*Technological Spaces: an Initial Appraisal*", CoopIS, DOA'2002, Industrial track, 2002.

[Kurtev, 2005] Kurtev, I., "*Adaptability of Model Transformations*", PhD Thesis, University of Twente, 2005.

[Lassila & Swick, 1999] Lassila, O., Swick, R., R., "*Resource Description Framework (RDF) Model and Syntax Specification*", W3C Recommendation. [Online]. Available: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[Matheus, 2004] Matheus, C. J., "*SWRLp: An XML-Based SWRL Presentation Syntax*", In Proceedings of the 3rd International Workshop on Rules and Rule Markup Languages for the Semantic Web, Hiroshima, Japan, pp. 194-199.

[McBride, 2002] McBride, B., “*Jena: A Semantic Web Toolkit*”, IEEE Internet Computing, vol. 6, no. 6, pp. 55-59, 2002.

[Metzger, 2005] Metzger, A., "*A Systematic Look at Model Transformations*". Showed in: Beydeda, S., Book, M., Gruhn, V., (eds.) "*Model-Driven Software Development*", Springer-Verlag, pp. 19.-33., 2005.

[Mens & Van Gorp, 2005] Mens, T., Van Gorp, P., “*A Taxonomy of Model Transformation*”. In Proceedings of the International Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, 2005.

[Miller & Mukerji, 2003] Miller, J., Mukerji, J., (eds.), "*MDA Guide Version 1.0.1*", OMG, 2003.

[Mizoguchi et al., 1995] Mizoguchi, R., Welkenhuysen, V., J., Ikeda, M., "*Task ontology for reuse of problem-solving knowledge*". In N.J.I. Mars, (ed.), Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95), Twente, The Netherlands, pp. 46.-57., Amsterdam, NL., IOS Press, 1995.

[MOF, 2002] "*Meta Object Facility (MOF) 1.4 Specification*", OMG Specification, ptc/02-04-02, 2002. [Online]. Available: http://www.omg.org/docs/formal/02-04-03.pdf.

[MOF, 2004] "*UML Profile for Metaobject Facility (MOF) Specification*", OMG adopted specification, February 2004, Version 1.0, formal/04-02-06. [Online]. Available: http://www.omg.org/docs/formal/04-02-06.pdf.

[MOF, 2005] "*Meta Object Facility (MOF) 2.0 Core Specification*", OMG Available Specification, ptc/04-10-15, 2005. [Online]. Available: http://www.omg.org/docs/ptc/04-10-15.pdf.

[Noy & McGuinness, 2001] Noy, N., McGuinness, L., D., "*Ontology Development 101: A Guide to Creating Your First Ontology*", Technical Report KSL-01-05, Knowledge Systems Laboratory, March 2001.

[OCL, 2006] "*Object Constraint Language*", OMG Available Specification, Version 2.0, formal/06-05-01, maj 2006.

[ODM, 2006] "*Ontology Definition Metamodel*", Sixth Revised Submission to OMG/ RFP, ad/2003-03-40, 2006. [Online]. Available: http:// http://www.itee.uq.edu.au/~colomb/Papers/ODM20060606.pdf

[Omelayenko & Fensel, 2001] Omelayenko, B., Fensel, D., "*A Two-Layered Integration Approach for Product Information in B2B Ecommerce*", In: K. Bauknecht, S. -K. Madria, G. Pernul (eds.), Electronic Commerce and Web Technologies, Proceedings of the 2nd Int. Conference on Electronic Commerce and Web Technologies, Germany, 2001.

[Parr, 2005] Parr, T., "*An Introduction To ANTLR*", [Online]. Available: http:// http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html.

[Passin, 2004] Passin, B., T., "*Explorer’s Guide to the Semantic Web*", Manning publications, 2004.

[Pilone & Pitman, 2005] Pilone, D., Pitman, N., "*UML 2.0 in a Nutshell*", O'Reilly, 2005.

[RFML, 2006] Relational-Functional Markup Language (RFML), http://www.relfun.org/rfml/, 2006.

[Roy & Ramanujan, 2001] Roy J., Ramanujan, A., "*XML Schema Language: Taking XML to the Next Level*", IEEE IT Professional, Vol. 3, No. 2, 2001.

[Seidewitz, 2003] Seidewitz, E., "*What Models Mean*", IEEE Software, pp. 26.-32., 2003.

[Selic, 2003] Selic, B., "*The Pragmatics of Model-Driven Development*", IEEE Software (spec. issue on Model Driven Development), Vol. 20, No. 5, 2003, pp. 19.-25.

[Sendall & Kozaczynski, 2003] Sendall, S., Kozaczynski, W., "*Model Transformation: The Heart and Soul of Model-Driven Software Development*", IEEE Software, pp. 42.-45., 2003.

[Sheth, 2006] Sheth, A., Verma, K., Gomadam, K., "*Semantics to energize the full services spectrum*", Comm. of the ACM, vol. 49, no. 7, pp. 55-61, 2006.

[Simons, 2005] Simons, A. J. H., "*The Theory of Classification, Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts*", in Journal of Object Technology, vol. 4, no. 2, March-April 2005, pp 15-26.

[Starfield et al., 1990] Starfield, M., Smith, K., A., Bleloch, A., L., "*How to model it: Problem Solving for the Computer Age*", McGraw-Hill, New York, 1990.

[SQL, 1999] "*Structured Query Language (SQL) standard*", InterNational Commitee for Information Technology Standards (INCITS), American National Standard ANSI/ISO/IEC 9075-2:1999, September 1999.

[Taentzer, 2006] Taentzer, G., "*Towards Generating Domain-Specific Model Editors with Complex Editing Commands*", In Proceedings of International Workshop Eclipse Technology eXchange(eTX), 2006.

[Taveter & Wagner, 2001] Taveter, K., Wagner, G., "*Agent-Oriented Enterprise Modeling Based on Business Rules*", in Proceedings of 20th International Conference of Conceptual Modeling (ER2001), Springer-Verlag, LNCS 2224, pp. 527.-540., 2001.

[Uhl & Ambler, 2003] Uhl, A., Ambler, W., S., "*Point/Counterpoint: Model Driven Architecture Is Ready for Prime Time / Agile Model Driven Development Is Good Enough*", IEEE Software, Vol. 20, No. 5, pp 70-73, 2003.

[UML, 2004] "*Unified Modeling Language (UML) Specification: Infrastructure*", version 2.0, Finalized Convenience Document, ptc/04-10-14, OMG, 2004. [Online]. Available: http:// http://www.omg.org/docs/ptc/04-10-14.pdf.

[UML, 2005] "*Unified Modeling Language: Superstructure*", version 2.0, formal/05-07-04, OMG specification, 2005. [Online]. Available: http://www.omg.org/docs/formal/05-07-04.pdf.

[Unhelkar & Henderson-Sellers, 2004] Unhelkar, B., Henderson-Sellers, B., "*Modelling Spaces and the UML*" Proceedings of the IRMA (Information Resource Management Association) Conference, New Orleans, 2004.

[W3C, 2000] "*Semantic Web Development: Technical Proposal*". [Online]. Available: www.w3.org/2000/01/sw/DevelopmentProposal.

[W3C, 2003] "*Semantic Web Activity Statement*". [Online]. Available: http://www.w3.org/2001/sw/Activity.

[Wagner, 2003] Wagner, G., "*Web Rules Need Two Kind Of Negations*", in Proceedings of Principles and Practice of Semantic Web Reasoning, PPSWR, pp. 33.-50., 2003.

[Wagner et al., 2003] Wagner, G., Tabet, S., Boley, H., "*MOF-RuleML: The Abstract Syntax of RuleML as a MOF model*", Integrate 2003, OMG Meeting, Boston, October 2003.

[Wagner et al., 2005] Wagner, G., Viegas Damásio, C., and Antoniou, G., "*Towards a general web rule language*", International Journal of Web Engineering and Technology, Vol. 2, Nos. 2/3, pp. 181-206, 2005.

[Wagner et al., 2006] Wagner, G., Giurca, A., Lukichev, S., "*A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL*", WWW2006, Edinburgh, UK, May 2006.

[Wagner et al., 2006-1] Wagner, G., Giurca, A., Lukichev, S., "*A General Markup Framework for Integrity and Derivation Rules*", Dagstuhl Seminar Proceedings, Principles and Practices of Semantic Web Reasoning, 2006. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2006/479.

[Wimmer & Kramler, 2005] Wimmer, M., Kramler, G., "*Bridging Grammarware and Modelware*", Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops, Montego Bay, Jamaica, October 2-7, pp. 159.-168., 2005.

[XMI2, 2005] "*MOF 2.0/XMI Mapping Specification, v2.*1", formal/05-09-01, OMG, 2005. [Online]. Available: http://www.omg.org/docs/formal/05-09-01.pdf.

[XML, 2006] XML meta-model, netBeans.org, [Online]. Available: http://mdr.netbeans.org/mdrxml.html.

[QVT, 2005] "*Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*", OMG Final Adopted Specification, ptc/05-11-01, 2005. [Online]. Available: http://www.omg.org/docs/ptc/05-11-01.pdf.

[QVT RFP, 2005] OMG, Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10), OMG Document ad/2005-07-01, 2005.