Study of operators for meta model composition and their implementation

Master report presented to
Centre Universitaire d'Informatique (CUI)
Geneva University

by

Xavier Eduardo Monnet

Supervisors:
Prof. Dr. Didier Buchs
Assistant Luis Pedro

Geneva, October 2008

Acknowledgements

In Memory Of My Grandmother.

The project started on October 2007 and ended on October 2008. A year of work on this master thesis.

My parents have been supporting me during all the project, and I would never have succeeded without their love, their help and their implication for helping me to keep my objectives clear and to push me to keep my hard work.

The main reason for having taken such a long time is because I have been working half time during my studies.

I want then to thank my ex-employer, Young and Rubicam, a world known advertisment agency, for having accepted to let me have two days per week for working on my Master and the University of Geneva for having accepted to let me take my time for finishing my thesis.

On the academic side, I want to give my thanks to my Profesor, Dr Didier Buchs, and to my supervisor Luis Pedro for their help and support during this project.

I also want to thank Sergio Coelho, a great friend that have been there in the good moments and in the bad ones that have occured in the past year.

And also thanks to Riccardo Gambone, Mathieu Duc, Fares-Hosni Naal, and Jean-Pierre Bernini for having offered me their help or support in different key moments.

A special thought to all my friends of the University and to the great people I have been working with at Young and Rubicam, without forgetting all the people that have contributed in different ways and different degrees to push me to keep moving forward and to evolve.

Abstract

The aim of this master's project is to study and implement operators for meta model composition. The meta-models are expressed in Eclipse Modeling Framework (EMF) [1] Ecore formalism and we should be able to compose them in order to get as result composed meta models.

The objective is also to be able to compose bricks of domain concepts of computer languages, as assignments, conditions or iterations, for getting a composed meta model that will describe a new language.

Another part of the work, is to study the composition of the instances of the meta model we are going to compose.

And in order to rely that with the aim of verification and testing, we also have defined transformations of those domain concepts into Concurrent Object Oriented Petri Nets (COOPN) [2] specifications, which is an object-oriented specification language based on synchronized algebraic Petri nets. This language allows the definition of active concurrent objects, and includes facilities for sub-typing, sub-classing and genericity. It is used for testing and verification.

This work is a part of the Model Transformation for Verification (MTV) [3] project which is being developed by the Software Modeling and Verification (SMV) [4] group at the University of Geneva.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main objective of this Master's Project is to study and implement operators for meta model composition.

In this chapter some concepts and keywords that are useful to understand the work done during this Master's Project are introduced.

1.1 Model-Driven Architecture

The Model-Driven Architecture (MDA) [5] is a software design approach, proposed and sponsored by the Object Management Group (OMG) [6]. A lot of different software design techniques have been proposed since the first steps of the software engineering. So, the question we can ask ourselves is: why is the OMG proposing MDA ?

The computer technologies progress and evolve in a very fast way. The work of the engineers often consist in adapting projects developed months ago with technologies that are not actual anymore. To solve this problem and to try to develop projects without being dependent on the technologies evolutions, MDA has been proposed. In fact MDA allows to separate design from architecture and realization technologies, facilitating that design and architecture can evolve independently.

The general idea of MDA methodology is to model at a high level of abstraction the logical and behavioral functionalities of a system after having specified the clients requirements in a Computation Independent Model (CIM). This high level of abstraction is called Platform Independent Model (PIM) and its main objective is to allow the specified functional requirements to survive to the changes done by the realization technologies and architectures. We can then transform the PIM to a Platform Specific Model (PSM). The huge interest of this technique is that this transformation can be performed automatically.

Some transformation softwares with the purpose of supporting MDA philosophy are being developed - the main goal is to allow that the ideas of specifying at a high-level of abstraction and then automatically generate different realizations of them can be a reality. Some of the most known softwares that support the MDA approach are, among others, MetaEdit+ [7], Generic Modeling Environment [8], ParadigmPlus [9] and DOmain Modeling Environment [10]. The MTV is also one of the framework that supports MDA approach although with the main goal of using the specification at a high-level of abstraction in order to be able to perform transformations specifically for software verification.

We can point that MDA can be seen as a good solution because when evolution of technologies occurs (and usually the occur very often), we should just need to regenerate the new PSM that respects the new technology. In principle, this makes a great economy of time.

Another thing I can add about MDA after interesting myself on reading what was written by specialized magazines, is that the MDA is seen as a great solution for the future of the software engineering. But some people seems to be not filled with enthusiasm. Some of their arguments are that MDA requires to much qualifications and knowledges. It is a fact that the development teams that will adopt MDA methodology will have to control concepts and languages as Unified Modeling Language (UML), Object Constraint Language (OCL), CIM, PIM, etc... But is this really a problem ? I don't think so. In the informatics world, one of the principal rule is that people are always having to learn new concepts, new technologies... that is nothing new. Moreover I will say that this is the reason why computer science makes such extraordinary progresses in a such fast way.

1.2 Of Models and Meta Models

A model is an abstraction of phenomena in the real world. In other words, it is a theoretical construct that represents some processes or behaviors of a given system, with a set of variables and a set of logical and quantitative relationships between them. There are a lot of modeling languages. Each one have his own advantages and defects. Some are formal but harder to use, other easier to understand but having ambiguity. For examples, Petri Nets, Z or OCL are modeling languages. Actually, the more used and accepted in the software development world is the Unified Modeling Language (UML) of the OMG. Modeling a system is the next logical step after producing the requirements documents. Knowing that approximately sixty per cent of the part of a software development budget is alloted to software's maintainability (corrective, adaptive and perfective) [11] we can easily understand why good modeling techniques and the Model-Driven Architecture (MDA) approach are fundamental.

A meta model is yet another abstraction, highlighting properties of the model itself. A model is said to conform to its metamodel like a program conforms to the grammar of the programming language in which it is written. In other words, a meta model is a collection of ”concepts” (things, terms, etc.) within a certain domain. The OMG usually uses following modeling architecture to present and explain the meta models the (also shown in Fig. 1.1) :

M0 level: Data layer contains usually source code derived from the *M1* level fro different program languages;

M1 level: Model layer, describes the information in *M0* layer. In other words, contains the models from which the source code in *M0* layer is generated. The models available in this layer can be transformed in several different source code languages, depending only on the fact if the languages support the concepts presented in the model;

M2 level: Meta model layer that is a description of the models in layer *M1*. It is this meta model that defines the abstract syntax of a language and details what artifacts can be present in the model;

M3 level: Meta meta-model layer are Meta Object Facilities (MOF) Ecore themselves. In this approach the assumption is that MOF is self-described. This means that we don't need need another level for MOF description. In summary, the *M3* layer is the description of the Meta model, i.e. the *M2* layer.

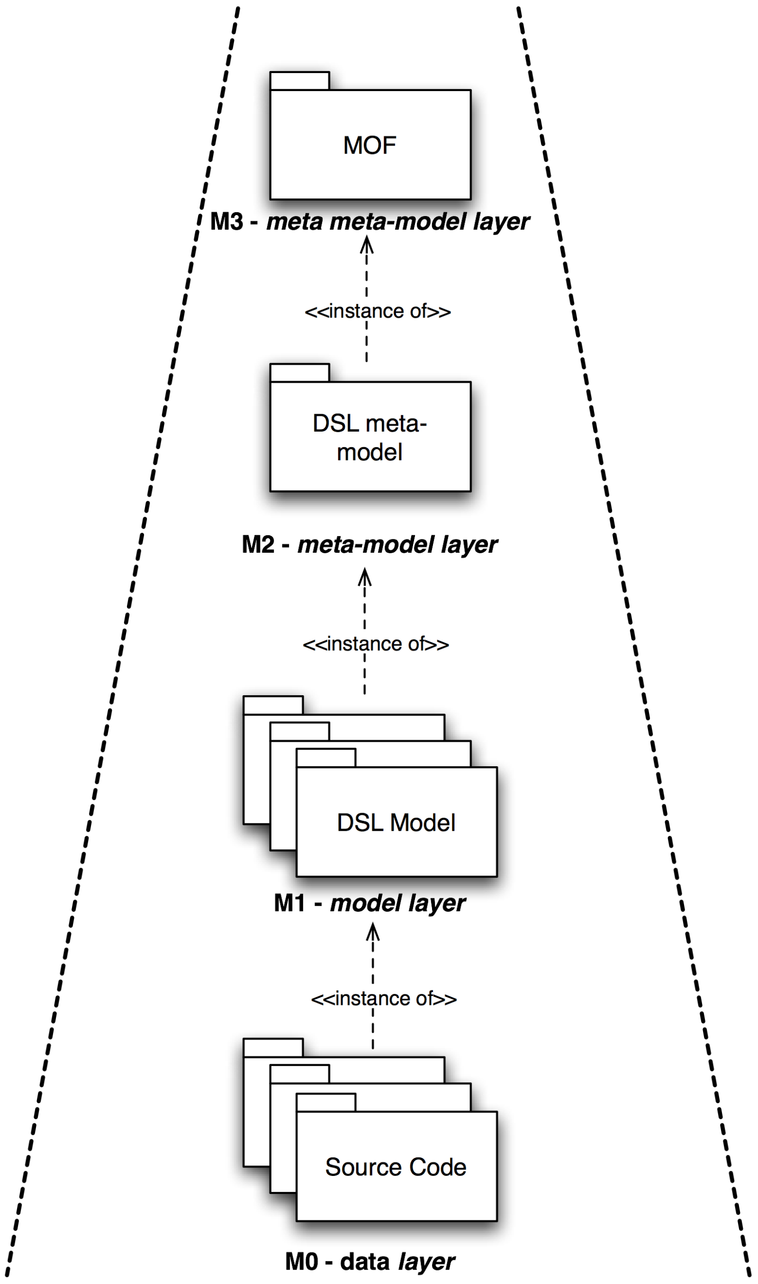


Figure 1.1. Architecture - Meta Object Facilities

The meta meta models are one more step of abstraction. The meta meta model describes the meta model in the same way that the meta model describes a model. The particularity consists on the fact that if we try to make another step in the abstraction, we will find the same meta meta model. This is why the meta meta models are said to be an auto-descriptive layer: the meta meta model is self described.

There are different approaches and implementations for using meta modeling techniques. One of them is the Meta Object Facilities (MOF) [12] supported by the OMG. Another one is the Eclipse Modeling Framework (EMF) Ecore [1] that is proposed by Eclipse project. We will introduce Ecore that is the meta model used by EMF in the next section.

As an illustration of this methodology take table 1.1

Table 1.1. Meta Modeling Architecture Layers with some examples

Meta-Level	**Description**	**Examples**
M0	Data / instances	Records in a DB table, instances of Java classes (abc -instance of java.lang.String)
M1	Metadata / models	Tables, columns in a database (records in a system catalog), concrete classes, methods, fields in Java program (java.lang.String instance of Java Class language construct) classes (abc -instance of java.lang.String)
M2	Meta-Metadata/ metamodels/ languages	Description of database (definition of things like Table, Schema, Column, etc.), description of language constructs (definition of Class, its attributes, contained elements methods, fields, etc.)
M3	Meta-metamodel	MOF or Ecore

For who might be interested on going deeper and reading more about meta models and MOF, a good work has been developed by Stephane Heck in his Bachelor's report [3] from the University of Geneva. He writes about the MOF architecture and gives a very interesting example in which he starts with a model and climb trough the levels of the modeling architecture showed above.

1.3 Eclipse Modeling Framework and Ecore

The EMF [1] [13] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XML Metadata Interchange (XMI), EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

The core EMF framework includes a meta model (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective Application Programming Interface (API) for manipulating EMF objects generically.

Ecore is also a meta meta model because it is its own meta model, in other words it describes itself. In Fig. 1.2 it is possible to see a simplified representation of the Ecore meta model.

An instance of an Ecore metamodel will be a model that is composed by ENamedElement objects. Some important elements are the EPackage, Eclassifier and the ETyped Elements. We see that all the concepts needed for modeling are described in Ecore.

1.4 Composition of Meta Models

We have seen on previous sections that we can have models and meta models. The question is now why meta model composition? But before we answer this question, we should explain what meta model composition means.

Given two meta models and a composition operator as inputs, a meta model composition will produce as output a new composed meta model.

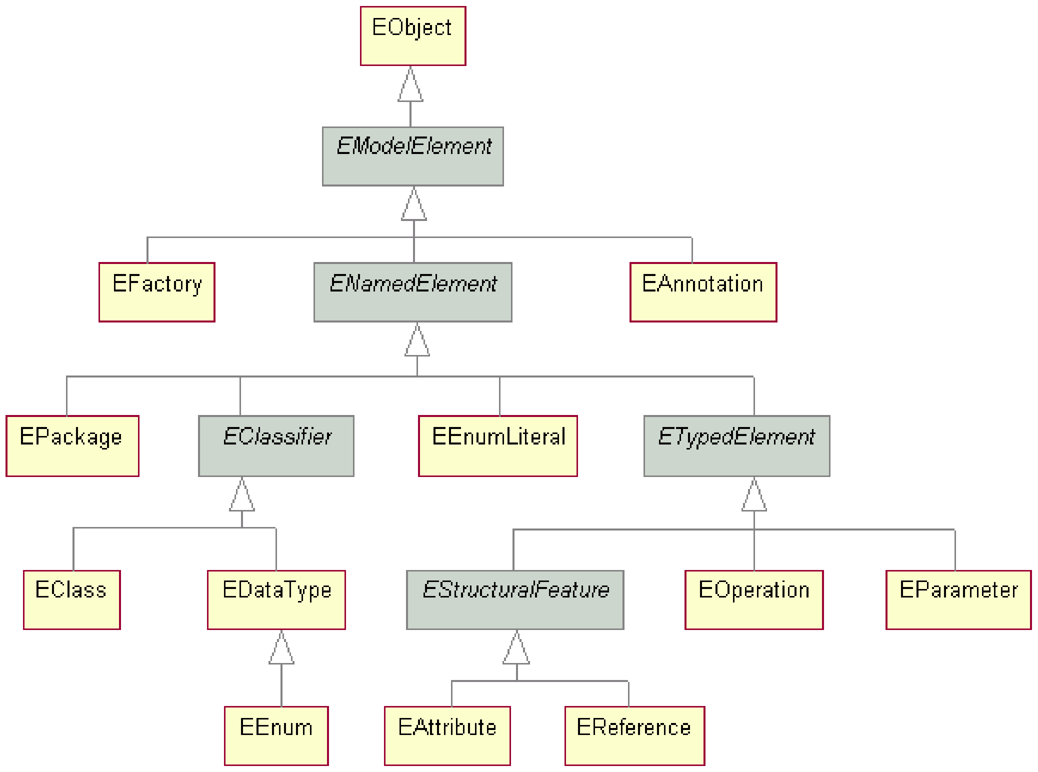


Figure 1.2. Simplified Ecore meta model

The function applied for generating this new meta model will depend on the selected composition operator. The description of what is done by each implemented operator (union, merge, association, aggregation and inherit) can be found on Chapter 3.3.

Why are we then interested on doing a meta model composition? That allows to define meta models that defines bricks of domain concepts (as assignment, condition, or iteration) and then to compose those bricks for generating new languages. It can also be useful for refactoring of meta models.

The another interesting question is what happen then with the instances of the input meta models? We need to define and implement the composition of the models. That means that we need to make the models (instances of meta models) conforms to the newly created meta model and to do an union of those models. Without forgetting to apply the changes depending on the composition operator. This question is presented in Chapter 6.

What is also interesting if we have the transformation of those domain concepts bricks into COOPN is to see what will happen then with the composed meta model. In ideal case, if we compose the transformations into COOPN, we will then be able to get a COOPN specification for the composed meta model and be able to proceed to test and verifications.

In following chapters you will find all these subjects aborded with more details.

Chapter 2

Description of the project

The main goal of this project is to study and implement operators for meta model composition.

The meta models used should conform to the EMF Ecore meta model. A tool for defining a composition specification and execute it must be done. The tool must also do the composition of the instances of meta models (composition of models). A study on transformation of domain concepts to COOPN is also required.

2.1 Project Phases

Here are the different phases that were identified before the start of the project :

- 1) Implement transformation from XMI based description of COOPN specification to COOPN specification;
- 2) Redefinition of the meta model for meta model composition;
- 3) Implement transformation of Assignment, Condition and Iteration Domain Concept (DC) to COOPN specifications;
- 4) Implement the meta model composition operators for meta models transformation;
- 5) Implement the transformation of instances of meta model for meta model composition;

- 6) Writing of the Master Project report;

The first step is to implement a tool for transformation an XMI based COOPN specification to a real COOPN specification.

Doing the second step (definition of meta model composition meta model) will allow us to generate with Eclipse an editor for creating composition specifications. For example, using this editor, we will be able to create an instance of the meta model composition meta model that will contain the union operator and two meta models paths.

The third step is some work on the domain concepts and their transformation to COOPN. This point should show how test and verification with COOPN can be done with meta model composition. The work it contains is:

- Defining the 3 meta models describing the domain concepts of Assignment, Condition and Iteration;
- Defining the corresponding COOPN specifications;
- Implementing the transformations from meta models to COOPN specifications.

The step four is working on the tool that will realize the composition of meta models. This means:

- Defining the 5 operators (union, merge, inherit, association, aggregation);
- Implementing examples for each operators with ATLAS Transformation Language (ATL) [14] model transformation language;
- Generalizing the transformations extracting ATL templates (one per operator);
- Implementing the tool for reading the composition specification and execute the corresponding automatically generated ATL transformation;

The fifth step is implementing the tool for model (instances of meta models) composition. This means that we need to conform the models to the new generated meta model and then to do an union of them applying the corresponding composition operator modifications to it.

Chapter 3

Transformation from XMI to COOPN

This chapter presents the work done for the implementation of a tool for transforming an XMI COOPN specification description into a COOPN specification launchable in the COOPNBuilder tool. The first section is a small introduction to COOPN language and the second section is about the transformation.

3.1 About COOPN Language

COOPN [2] is an object-oriented specification language based on synchronized algebraic Petri nets. CoopnBuilder is an integrated development environment designated to the support of concurrent software development based on the COOPN language. Briefly, COOPN defines Petri Nets (PN) [15] and coordination between PN using object-oriented approach.

The main characteristics of COOPN are:

- Declarative language: using prolog-like syntax, behavior of PN are defined by rules. Transitions are specified with pre- and post- conditions;
- Algebraic Petri net based: the essential elements of a COOPN defined system are place, transition(event) and token. They are extended notion of classical place/transition nets because of their richness and complicity. Its fairly easy to represent classical P/T nets and colored Petri nets using COOPN, the inverse is often not possible;
- Object-oriented: a Petri net is encapsulated by means of COOPN class, only defined methods/events are visible from exterior;

- Concurrency: the semantic of COOPN guarantees the synchronization of events of defined Petri nets, three primitive operators are: alternative, parallel and sequential;
- Coordination: COOPN context is a higher level of encapsulation, its an environment model and specifies the connections of internal components: objects (instances of classes) and sub-contexts. COOPN contexts do not define Petri nets but their compositions;

The execution of COOPN uses prolog-like interference engine.

COOPN can be used to model large, concurrent, transactional, distributed systems and construct prototypes for these systems. CoopnBuilder provides the facilities to edit, verify COOPN modules, compile them into java prototype and interpret, simulate the prototype.

There are three types of COOPN module: Abstract (Algebraic) Data Type (ADT), Object and Context.

ADT represents data with no states and they are immutable. ADTs have operations. Class has places, methods(visible transitions), non-visible transitions and gates(outgoing events). Places may contain ADT data or objects. Context has methods and gates. Methods are input events, gates are output events. Both of them can have parameters, the type of parameters can be ADT or COOPN class.

We define tokens using ADT with algebraic specifications, thus the type of tokens are richer than colored Petri nets, in addition, tokens support operations.

More information about COOPN can be found on the COOPN Blog [16] or on the SMV Website [4].

3.2 XMI COOPN Specification to COOPN Specification Transformation Tool

As for all transformation, we have an input and an expected output. In this case, as shown in Fig. 3.1, we have a COOPN specification described in an XMI format conforming to the COOPN language meta model and we want as output the corresponding COOPN specification that should be opened in COOPNBuilder in order to be able to generate the corresponding prototype and run it.

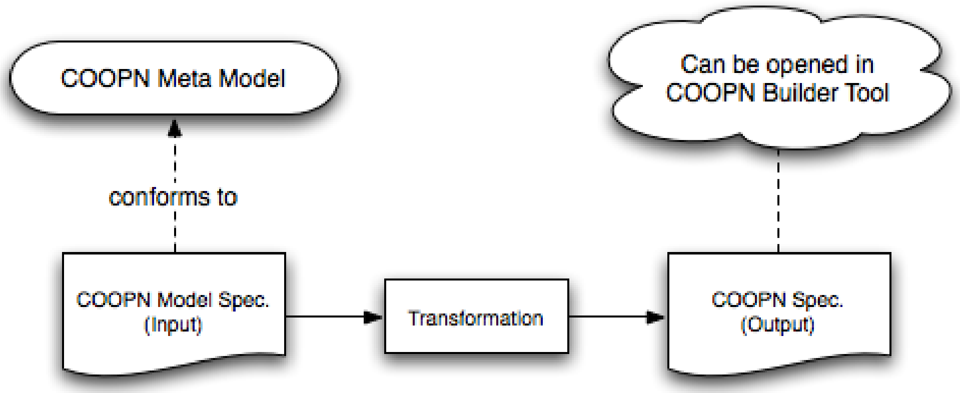


Figure 3.1. View of the transformation from XMI based description of a COOPN specification to its COOPN specification.

3.2.1 Input: XMI COOPN Specification

The input of this transformation is an instance of the meta model of the COOPN language.

The meta model for the COOPN language has been defined by Sergio Coelho in his master thesis. For more information about this meta model, please refer to his master's report [17].

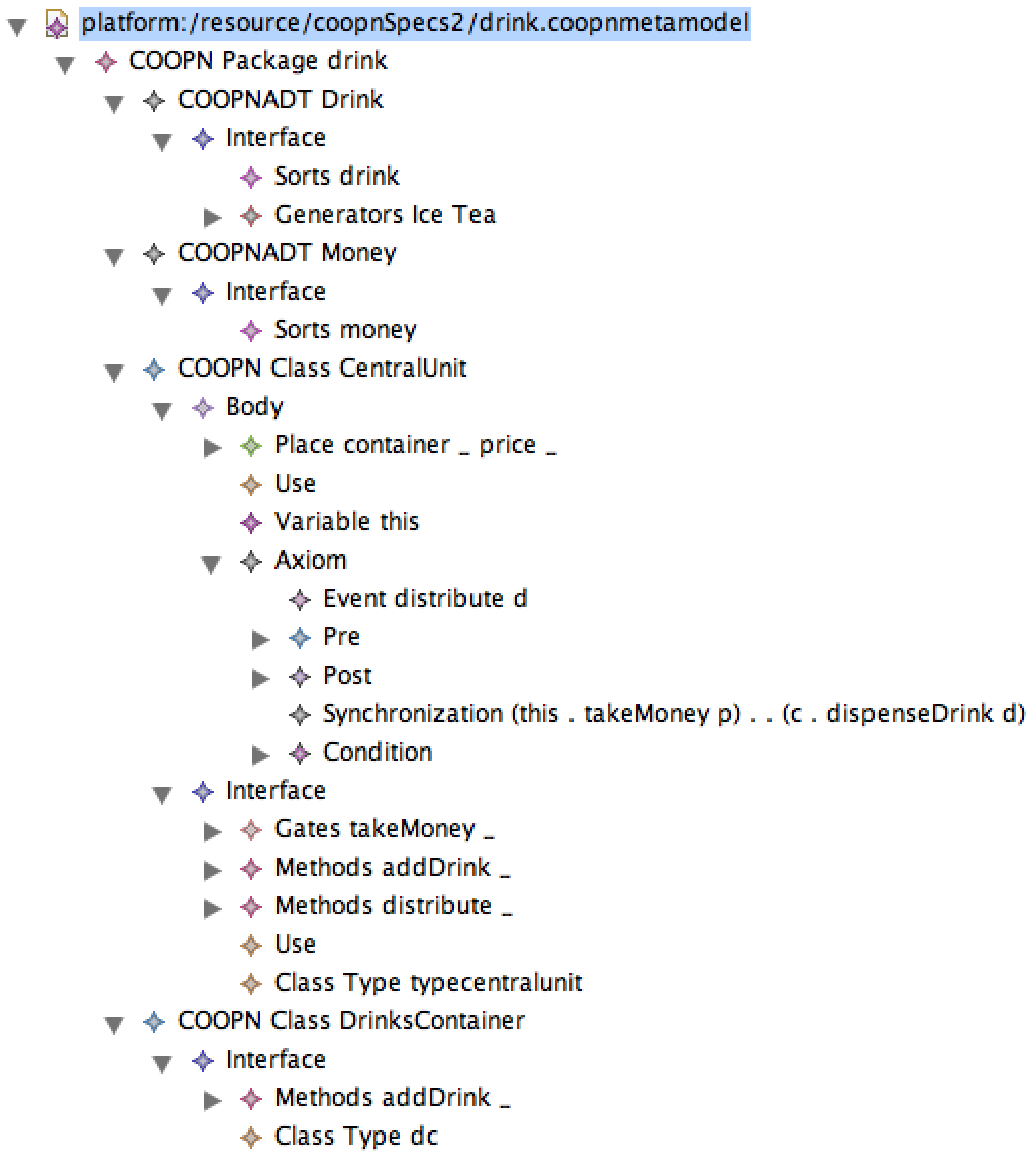
An instance of this meta model will contain elements as ADTs, Classes, or Context. Each one of these elements possess an Interface for all Methods we can call from the exterior and a Body for all the Places, Transitions, Axioms and Methods that are accessible only from the inside of the element. Detailed information about this can be found at SMV Group website.

You can see in Fig. 3.2 an example of an XMI COOPN Specification in a tree view.

3.2.2 Output: COOPN Specification

The output of this transformation is a COOPN Specification package that can be opened in the COOPNBuilder Tool.

An example of generated COOPN Specification opened the COOPN-Builder Tool can be seen in Fig. 3.3. It is an example of specification for a Drink Vending Machine.



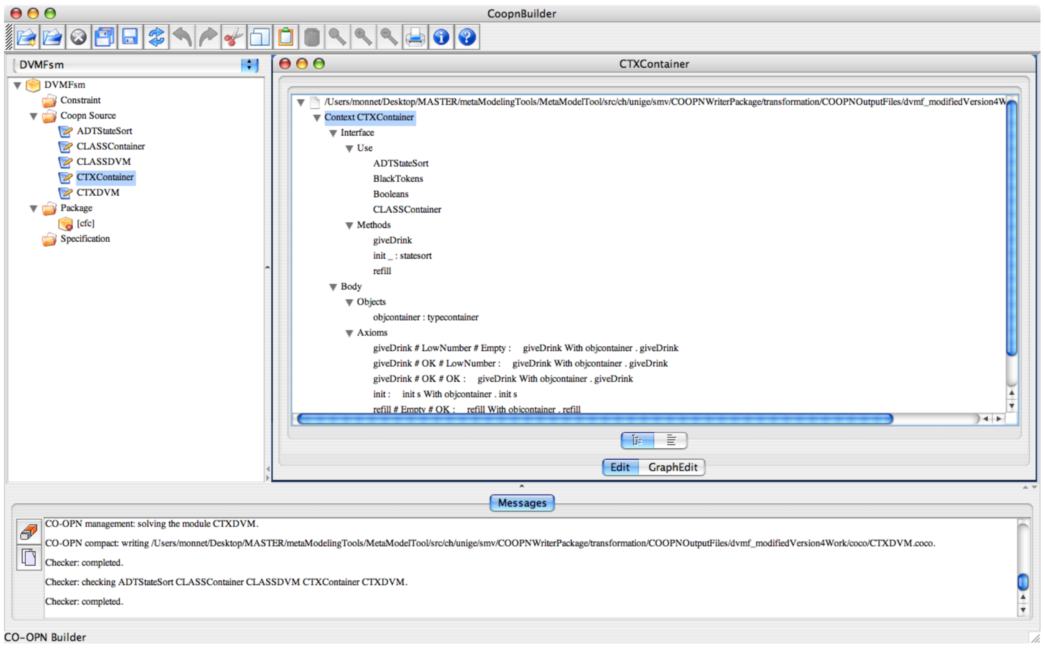
```
▼ platform:/resource/coopnSpecs2/drink.coopnmetamodel
    ▼ COOPN Package drink
        ▼ COOPNADT Drink
            ▼ Interface
                    Sorts drink
                ▶ Generators Ice Tea
        ▼ COOPNADT Money
            ▼ Interface
                    Sorts money
        ▼ COOPN Class CentralUnit
            ▼ Body
                ▶ Place container _ price _
                    Use
                    Variable this
                ▼ Axiom
                        Event distribute d
                    ▶ Pre
                    ▶ Post
                        Synchronization (this . takeMoney p) . . (c . dispenseDrink d)
                    ▶ Condition
            ▼ Interface
                ▶ Gates takeMoney _
                ▶ Methods addDrink _
                ▶ Methods distribute _
                    Use
                    Class Type typecentralunit
        ▼ COOPN Class DrinksContainer
            ▼ Interface
                ▶ Methods addDrink _
                    Class Type dc
```

Figure 3.2. Example of XMI COOPN Specification: the drinks example.

3.2.3 Transformation Implementation

The implementation language that have been chosen for that transformation is Java.

It is a one to one kind of transformation, as for each element of the



```
▼ DVMFsm
    Constraint
  ▼ Coopn Source
      ADTStateSort
      CLASSContainer
      CLASSDVM
      CTXContainer
      CTXDVM
  ▼ Package
      [cfc]
    Specification
```

```
▼ /Users/monnet/Desktop/MASTER/metaModelingTools/MetaModelTool/src/ch/unige/smv/COOPNWriterPackage/transformation/COOPNOutputFiles/dvmf_modifiedVersion4W
  ▼ Context CTXContainer
    ▼ Interface
      ▼ Use
          ADTStateSort
          BlackTokens
          Booleans
          CLASSContainer
      ▼ Methods
          giveDrink
          init _ : statesort
          refill
    ▼ Body
      ▼ Objects
          objcontainer : typecontainer
      ▼ Axioms
          giveDrink # LowNumber # Empty :    giveDrink With objcontainer . giveDrink
          giveDrink # OK # LowNumber :    giveDrink With objcontainer . giveDrink
          giveDrink # OK # OK :    giveDrink With objcontainer . giveDrink
          init :    init s With objcontainer . init s
          refill # Empty # OK :    refill With objcontainer . refill
```

```
CO-OPN management: solving the module CTXDVM.
CO-OPN compact: writing /Users/monnet/Desktop/MASTER/metaModelingTools/MetaModelTool/src/ch/unige/smv/COOPNWriterPackage/transformation/COOPNOutputFiles/dvmf_modifiedVersion4Work/coco/CTXDVM.coco.
Checker: completed.
Checker: checking ADTStateSort CLASSContainer CLASSDVM CTXContainer CTXDVM.
Checker: completed.
```

Figure 3.3. Example of COOPN Specification: the drink vending machine. Generated with the XMI to COOPN Tool and opened in the COOPNBuilderTool.

instance of the COOPN meta model we are going to create the equivalent element in COOPN language.

The Fig. 3.4 shows the class diagram for the tool.

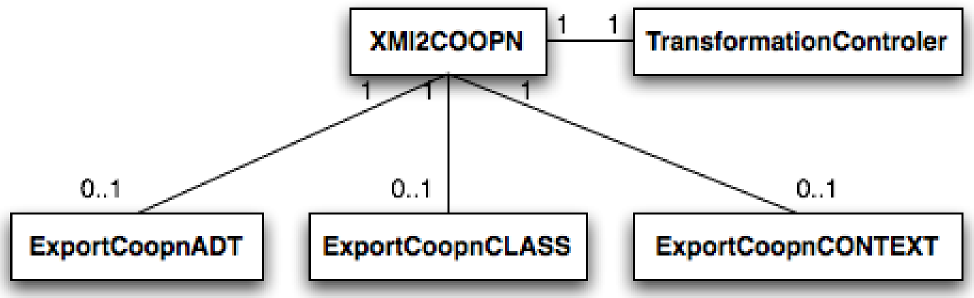


Figure 3.4. Class Diagram for XMI to COOPN Tool.

There a class called XMI2COOPN that uses:

- a TransformationController that allows to initialize the XMI resource set and register the type of XMI file to handle with (.ecore, .basic, .simple, etc), to initialize models and dependencies (in the EMF API) and to retrieve default Factories to create objects of these metamodels;
- an ExportCoopnADT for creations of ADTs;
- an ExportCoopnCLASS for creations of Classes;
- an ExportCoopnCONTEXT for creations of Contexts;

An interesting implementation issue was how to deal with the equations in the COOPN condition, axioms and theorems that are present in ADTs, Classes and Contexts.

You can find an example of expected axioms result in following listing.

Listing 3.1. Example of generated axioms with the recursive method.

```
Class CLASSContainer;

Interface
    Use
        ADTStateSort;
        BlackTokens;
        Booleans;
    Type
        typecontainer;
    Methods
        giveDrink;
        init _ : statesort;
        refill;

Body
    Places
        Empty _ : blackToken;
        LowNumber _ : blackToken;
        OK _ : blackToken;
        initialized _ : boolean;
    Initial
        initialized false;
    Axioms
        giveDrink::
                LowNumber @ -> Empty @;
        giveDrink::
                OK @ -> LowNumber @;
        giveDrink::
                OK @ -> OK @;
        refill::
                Empty @ -> OK @;
        ((this = Self) = true) & (s = Empty) =>
                init Empty::
                initialized false -> initialized true, Empty @;
        this = Self = true & s = LowNumber =>
                init LowNumber::
                initialized false -> initialized true, LowNumber
                    @;
        this = Self = true & s = OK =>
                init OK::
                initialized false -> initialized true, OK @;
    Where
        s : statesort;
        this : typecontainer;

End CLASSContainer;
```

The solution for implementing that has been to define a recursive method. The terminal case is if the terms of the expression are not equation. If one or both of the terms of an equation are equation themselves, we then proceed to a recursion calling again this method on the equation term. The code corresponding to the implementation of this can be found in Appendix.

A point that have been quite painfull when doing this transformation has been the fact that the meta model for the COOPN language has been changing. And in a lot of cases this has means to rework more than once parts of the transformation. In the same time, it is clear that working on this transformation has helped to perform the meta model as we were seeing things that were not working.

Chapter 4

Transformation of Domain Concepts to COOPN Specifications

This chapter presents how we have defined and transformed to COOPN [2] specifications the Assignment, the Condition and the Iteration domain concepts.

In the first section of this chapter are presented the defined meta models for the domain concepts taken into consideration.

In the second section, the correspondant COOPN specification for each domain concept is presented.

Finally, the last section is about the details of the transformation from the meta model to the COOPN specification.

Remind that information about the COOPN language can be found on previous chapter.

4.1 Domain Concepts Meta Models

This chapter first describes the meta models defined for the Assignment, Condition and Iteration domain concept. It will then present the expected COOPN result. And finally, it explains how we can transform these domain concepts into defined COOPN specifications.

4.1.1 Assignment Domain Concept Meta Model

An assignment is a domain concept, which is fundamental in computer programming. An assignment is used in order to set or reset the value stored in a storage location denoted by a variable name.

In Fig. 4.1 is the meta model done for the assignment domain concept.

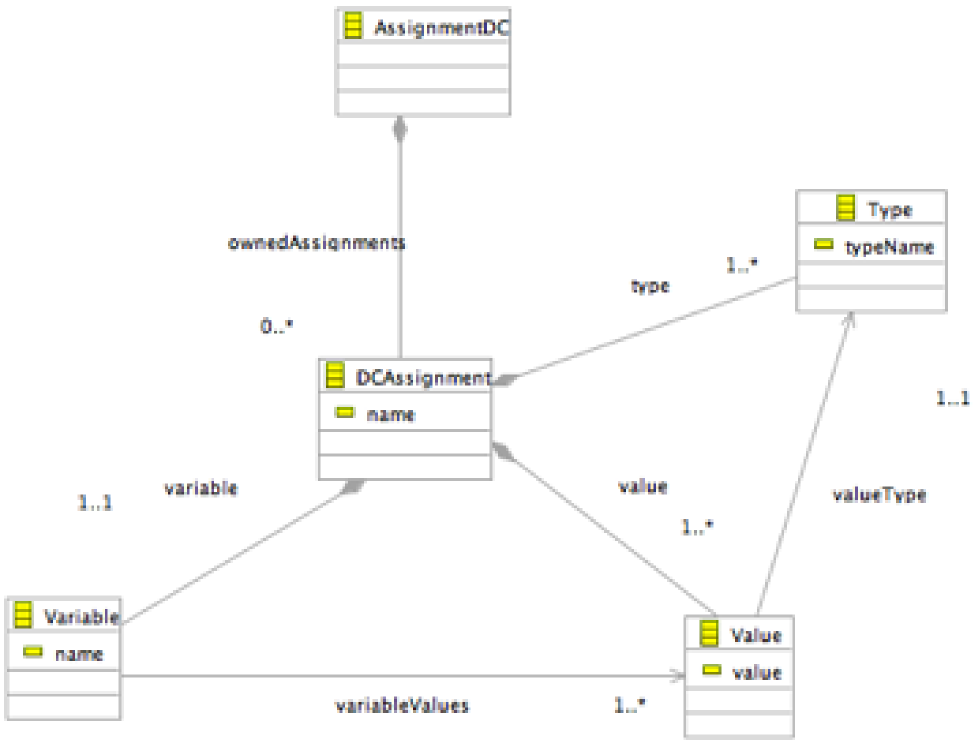


Figure 4.1. Assignment Domain Concept Meta Model

The AssignmentDC class stays as a container that contains all the assignments. It allows to describe more than one assignment in the same model.

The DCAssignment has a name that is used for being able to identify the different assignments. Each assignment is composed of a Variable (with a variable name), of at least one Value and of at least a Type.

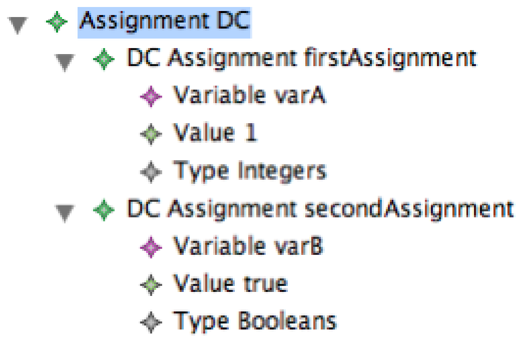
Each Variable must have at least one value, but more values can be associated. That is in order to deal with assignment of arrays for example.

Each Value must be associated to a Type.

Here is an example of a model that conforms to the Assignment meta model. It stands for the following assignments:

- varA = 1;
- varB = true;

Find in Fig. 4.2 the tree view of the assignment model used for this example.



```
▼ Assignment DC
  ▼ DC Assignment firstAssignment
      Variable varA
      Value 1
      Type Integers
  ▼ DC Assignment secondAssignment
      Variable varB
      Value true
      Type Booleans
```

Figure 4.2. Assignment Domain Concept Spec

4.1.2 Condition Domain Concept Meta Model

A condition is another fundamental concept for computer programming. A condition allows the execution of a block of instructions if the evaluation of a Boolean expression is true, or the execution of a different block of instructions in the case the evaluation of the Boolean expression is false.

In Fig. 4.3 is the meta model for the condition domain concept.

As for assignments, the ConditionDC allows us to store in the same model more than one condition.

The DCCondition has a name that is used for being able to identify the different conditions. A DCCondition is a specialization of Instruction, which means that a condition is an instruction. Each condition is composed of one BooleanExpression, of positive condition instructions (the instructions list to be executed if the Boolean expression is evaluated to true) and of negative condition instructions (the instructions list to be executed if the Boolean expression is evaluated to false). Both positive and negative lists of instructions can be empty.

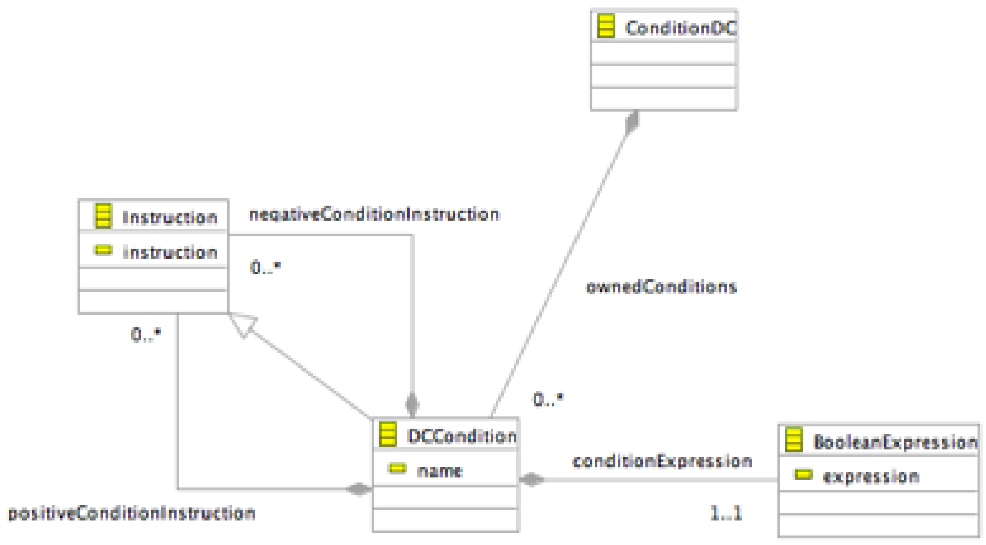
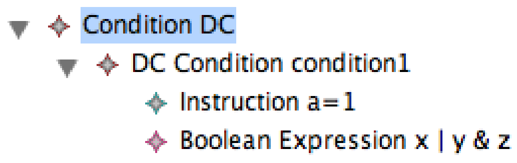


Figure 4.3. Condition Domain Concept Meta Model

Let's see an example of condition and it's resulting description in an instance of the defined meta model for the Condition Domain Concept presented before. Here is the condition instruction sample:

- if(x OR y AND z) then (a=1;);

In Fig. 4.4, you can see the corresponding instance of meta model that represents the model for the above example.



```
▼ Condition DC
    ▼ DC Condition condition1
          Instruction a=1
          Boolean Expression x | y & z
```

Figure 4.4. Condition Domain Concept Spec

4.1.3 Iteration Domain Concept Meta Model

An iteration is another a concept for computer programming. An iteration allows the execution of a block of instructions while the result of the evaluation of a Boolean expression is true. When the evaluation of the Boolean expression is false, the process stops.

In Fig. 4.5 is the meta model for the iteration domain concept.

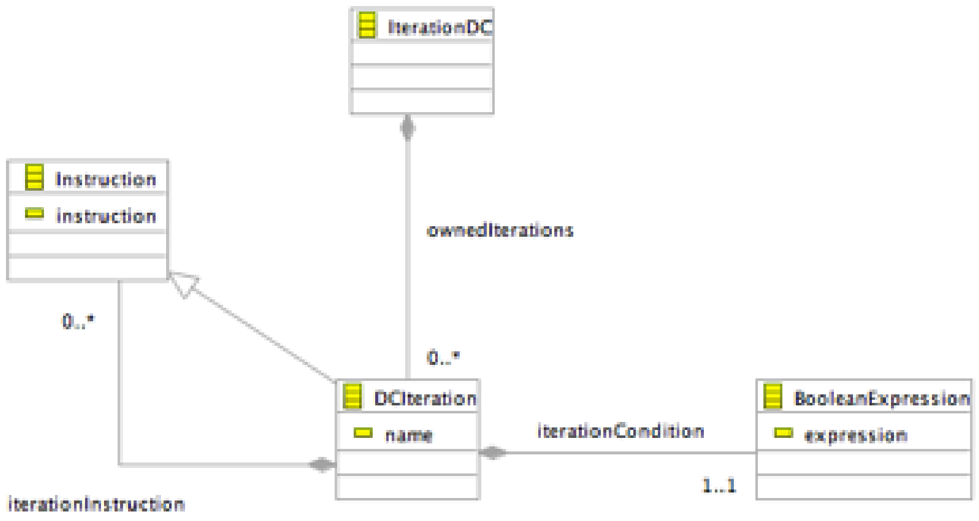


Figure 4.5. Iteration Domain Concept Meta Model

As for the other domain concepts, the IterationDC allows us to store in the same model more than one iteration concept.

The DCIteration has a name that is used for being able to identify the different iterations concepts. A DCIteration is a specialization of Instruction, which means that an iteration is an instruction. Each iteration concept is composed of one BooleanExpression and of Instructions. The instructions list is the instructions to be executed while the Boolean expression is evaluated to true. This list of instructions can be empty.

Let's define a small example of iteration and see its corresponding instance of meta model specification.

That is the example we are going to take into consideration:

- while(A smaller than 5) do (a++;);

In Fig. 4.6, you can see the corresponding instance of meta model that represents the model for the above example.

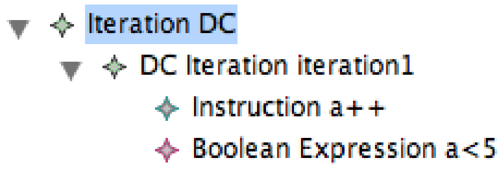

Iteration DC
DC Iteration iteration1
Instruction a++
Boolean Expression a<5

Figure 4.6. Iteration Domain Concept Spec

4.2 Domain Concepts COOPN Specifications

This section presents the results we are expecting into COOPN for each domain concept. It shows the COOPN expected result and the algorithms for generic transformations from domain concept to COOPN.

4.2.1 Assignment Domain Concept COOPN Specification

We have a meta model for Assignment Domain Concept with an instance of this meta model presented in a previous section.

In this section we are going to start by the presentation of what is the expected result into COOPN for the example presented in that previous section. After this is done, we will see the algorithm proposed to compute automatically the transformation. For doing the transformation we can do it implementing a Java module or using the Eclipse ATL [14] plug in for model transformations.

The expected COOPN specification description:

As seen in previous chapter, an assignment is the process of setting or resetting a value to a variable.

See Fig. 4.7.

We define a COOPN class called CLASSVariableRepository. This class will contain all the variables of the assignment described in the model.

Each variable will have a COOPN class place. The name of this place is the name of the correspondent variable. The content of the place will be the storage place for the value of the assigned variable. The type of the place is logically the type of the value that will be assigned to the variable.

Then we define a method in the interface of this class that will allow us to set a value to a variable, which means the method will put a token with the value of the corresponding variable to the place corresponding to

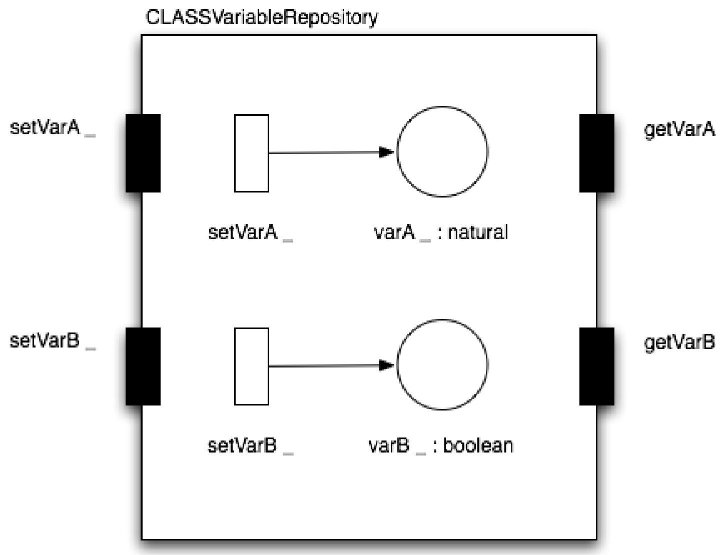


Figure 4.7. View of the expected COOPN Class Variable Repository for assignment domain concept

the variable. In a first attempt we tried to have one method for setting any place. But, as we were faced to some problems in COOPN , we decided finally to have one set method per variable. But here are the explanations for a more generalized set method. The signature of this method is:

- set - - : varnames vartypes ;

The varnames type is defined by a COOPN ADT that enumerate the names of the distinct variables of the model. The vartypes type is defined by a COOPN ADT that inherit from all the types used in the assignment. This allows us to have only one method for setting any place. For example, if we want to set the value true to the place varB, we will need to call the method set like this:

- set varB true ;

COOPN Axioms have been defined for defining the behavior of this method.

Then we have defined a COOPN Context that has a method execute-Assignment that is the method used to realize the assignment. When this method is called, it will execute all the assignments present in the assignment model.

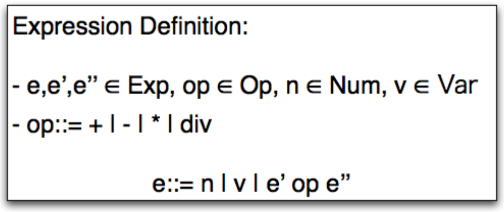
For example, if the model describes varA=1 and varB=true assignments, the executeAssignment method will set varA 1 and set varB true in order to set in the VariableRepository the values of the variables. This is done sequentially but could have done in parralel as there are now some parallel memories.

4.2.2 Condition Domain Concept COOPN Specification

As seen in previous sections with the Condition Domain Concept Meta Model, a condition contains three main elements:

- the boolean expression expressing the condition (the If statement);
- the instructions bloc to be executed in case the boolean expression is evaluated to true (the Then statement);
- the instructions bloc to be executed in case the boolean expression is evaluated to false (the Else statement);

We now need to define what a boolean expression is. For that, we first need to study what an expression is in programming languages. Assuming that we have Exp as the set of Expressions, Op the set of Operators, n as the set of Numbers constants and Var the set of Variables, we can define an Expression with the formula presented in Fig. 4.8.

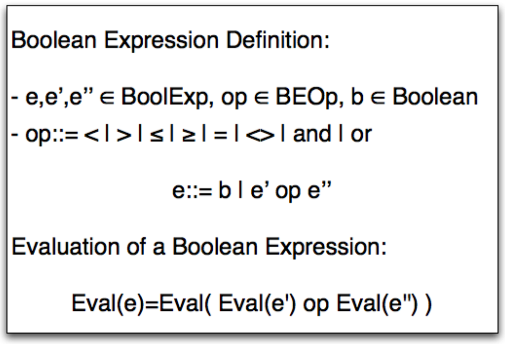

Expression Definition:

- e,e',e'' ∈ Exp, op ∈ Op, n ∈ Num, v ∈ Var
- op::= + I - I * I div

e::= n I v I e' op e''

Figure 4.8. Definition of an Expression

This helps to understand what an expression is, and is important if we want to define what a Boolean Expression is. Indeed, a Boolean Expression is a subsort of the Exp set. It has his own logical operators and can be define as in Fig. 4.9.



Boolean Expression Definition:

- e,e',e" ∈ BoolExp, op ∈ BEOp, b ∈ Boolean
- op::= < | > | ≤ | ≥ | = | <> | and | or

e::= b | e' op e"

Evaluation of a Boolean Expression:

Eval(e)=Eval(Eval(e') op Eval(e"))

Figure 4.9. Definition of a Boolean Expression and of its evaluation

In other words, a boolean expression can be:

- terminal case: a boolean value;
- a composition of boolean expressions with a logical operator;

An important point is the evaluation of the boolean expression which is defined in Fig. 4.9 and that explains that the evaluation a boolean expression is equivalent to the evaluation of the composition of the evaluation of each term of the expression.

So for example, if we want to evaluate the boolean expression (x or y and z), we will need to evaluate the values of the x,y,z variables. So in COOPN , we will have a CLASSVariableContainer as presented in previous section (assignment domain concept) containing the values of those boolean variables. So, we will have to getVarX, getVarY and getVarZ to be able to do the evaluation of the boolean expression. Then, the evaluate method will be able to perform the evaluation of (x or y) and then of (result(x or y) and z).

After having introduced some theorical concepts, lets see now how we can propose in COOPN an equivalent to the Condition Domain Concept Meta Model. In COOPN specification, we are then going to define for each condition the following COOPN elements:

- a Class for the boolean expression (*);

- a Class for the positive bloc of instructions (the Then bloc);
- a Class for the negative bloc of instructions (the Else bloc);
- a Class for managing the conditions;
- a Context;
- an ADT for listing the conditions names;

(*) In a first attempt, we thaught that we were needing a Class for the boolean expression, but in fact we don't need this element as we don't really need to know the state of the boolean expression. We just need a method in the ConditionManager for evaluation the Boolean Expression.

Let's describe each elements of this list.

The positive and negative instructions bloc objects have both the same structure. Each instruction of the bloc of instructions will be represented by an internal transition. So we will have a sequence of places and transitions that must be fired sequentially when the execute method of the Class is called. We need to add a transition and place for the begin and the end of the bloc, as we can have an empty bloc of instructions. The execute method will synchronize the Begin and End methods for verifiying that all has been working fine during the execution of the instruction. For the End method, we need to have a token in the last instruction place. See Fig. 4.10.

See Fig. 4.11 for an example of empty bloc of instructions.

The ConditionManager Class have a method for doing the evaluation of the Boolean Expression. It will do some getter methods for getting values of variables of the boolean expression in the CLASSVariableRepository and will do then the evaluation. The ConditionManager Class have also a method execute. This method takes as parameter the name of a condition (type defined with an ADT). This method does the following as depicted in Fig. 4.12.:

- it test the name of the condition for selecting the corresponding objects;
- it calls the eval boolean expression method, and gets the value of the evaluation;
- depending of the result of the evaluation, it will call the execute method of the positive or of the negative instruction bloc Class;

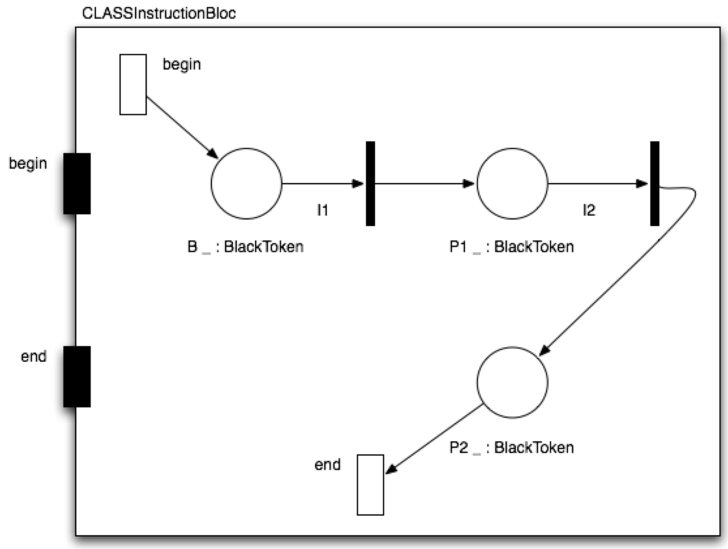


Figure 4.10. Graphic view of the COOPN Class for Instruction Bloc in Condition Domain Concept COOPN Specification - The transitions I1, I2, ..., In represents the n instructions of the instruction block and can be synchronized with other objects as for example with the ClassVariableRepository in order to execute an assignment.

The Context simply possess a method called doConditions that will execute in sequence the execute method for each condition.

4.2.3 Iteration Domain Concept COOPN Specification

For the Iteration Domain Concept, we can reuse notions and concepts defined for the assignment and the condition domain concepts. Indeed, as seen in previous sections with the Iteration Domain Concept Meta Model, an iteration contains two main elements:

- the boolean expression expressing the condition (the while conditon);
- the instruction bloc to be executed in case the boolean expression is evaluated to true (the do statement);

As for the condition domain concept, we have a boolean expression that should be evaluated. Then, we can reuse the evaluation method defined for condition domain concept.

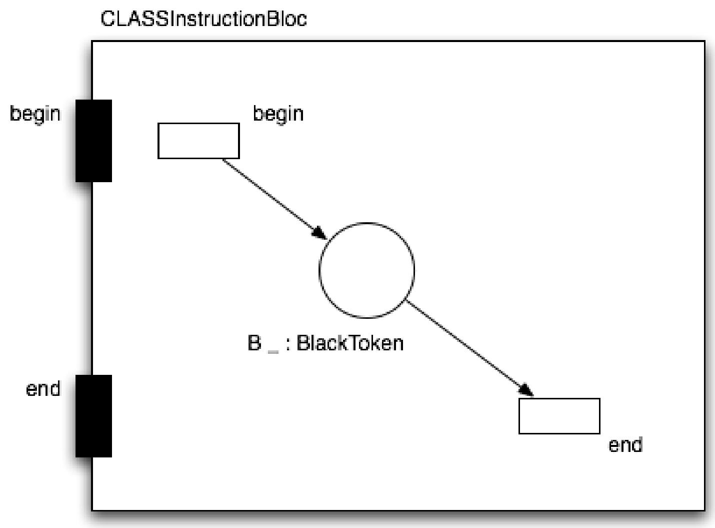


Figure 4.11. Graphic view of the COOPN Class for Instruction Bloc in Condition Domain Concept COOPN Specification - In this case, the instruction block contains no instruction. So we just have the begin and end methods.

Then, in case this result of this evaluation is true, we should execute a bloc of instructions. For that, we can also reuse the work done for instructions block for condition domain concept.

The small change concerns the IterationManager that should use a recurent execute method and the instructions could change the result of the evaluation. The execute iteration method works as follows:

- 1) evaluation of the iteration condition;
- 2.a) if evaluated to false, that's the stop condition;
- 2.b) if evaluated to true, we can execute the instruction bloc and sequentially call the iteration execute method again (that will start again to point 1);

The instruction bloc should modify the result of the evaluation if we want the iteration to have an end. For example, we can have a variable a initialized to 1, and do an iteration that will increment the value of the variable a untill the value of the a variable is less than 5. This is what is depicted in Fig. 4.13.

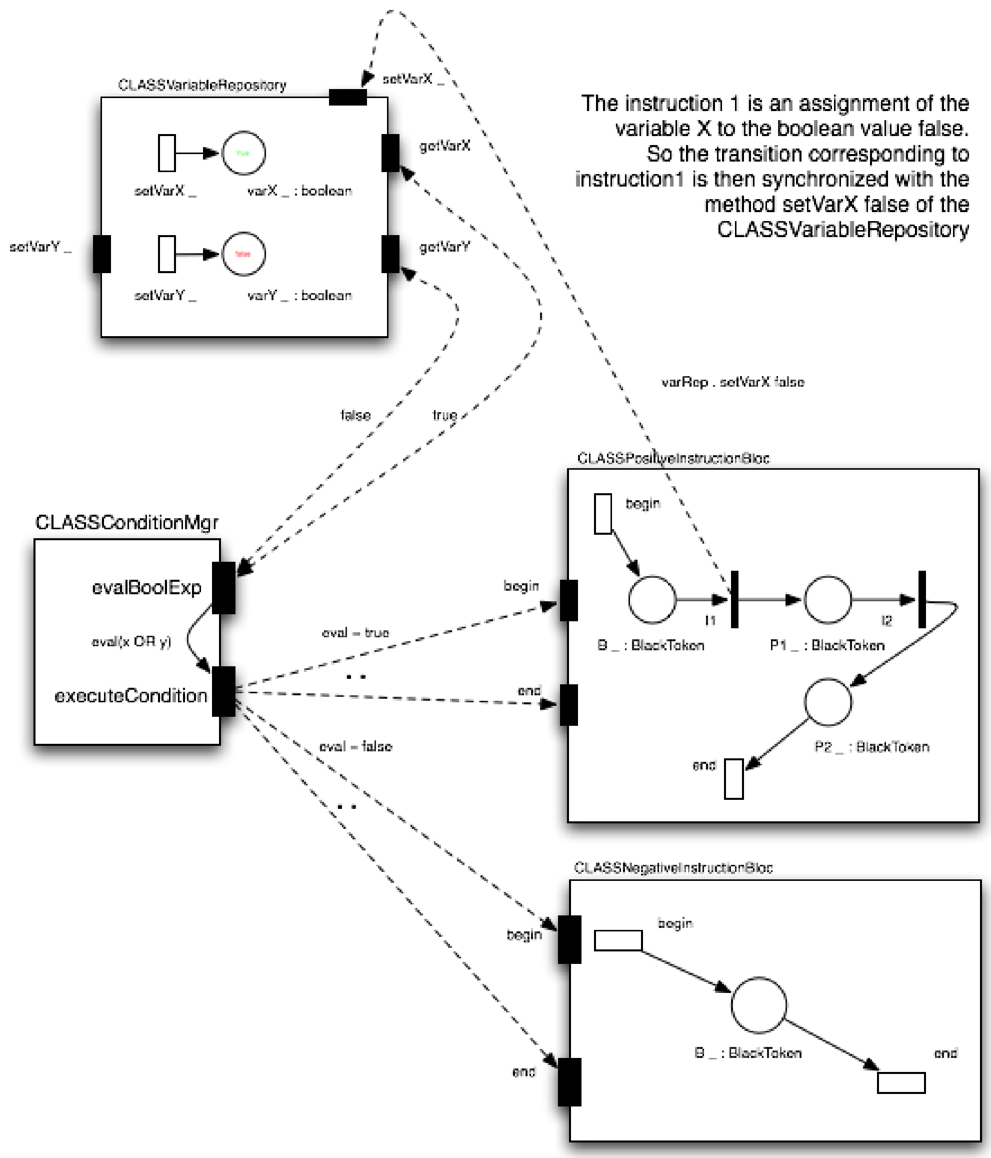


Figure 4.12. Graphic view of the COOPN Condition Manager Class and interactions with other objects. Example: if (x OR y) then (x=false;)

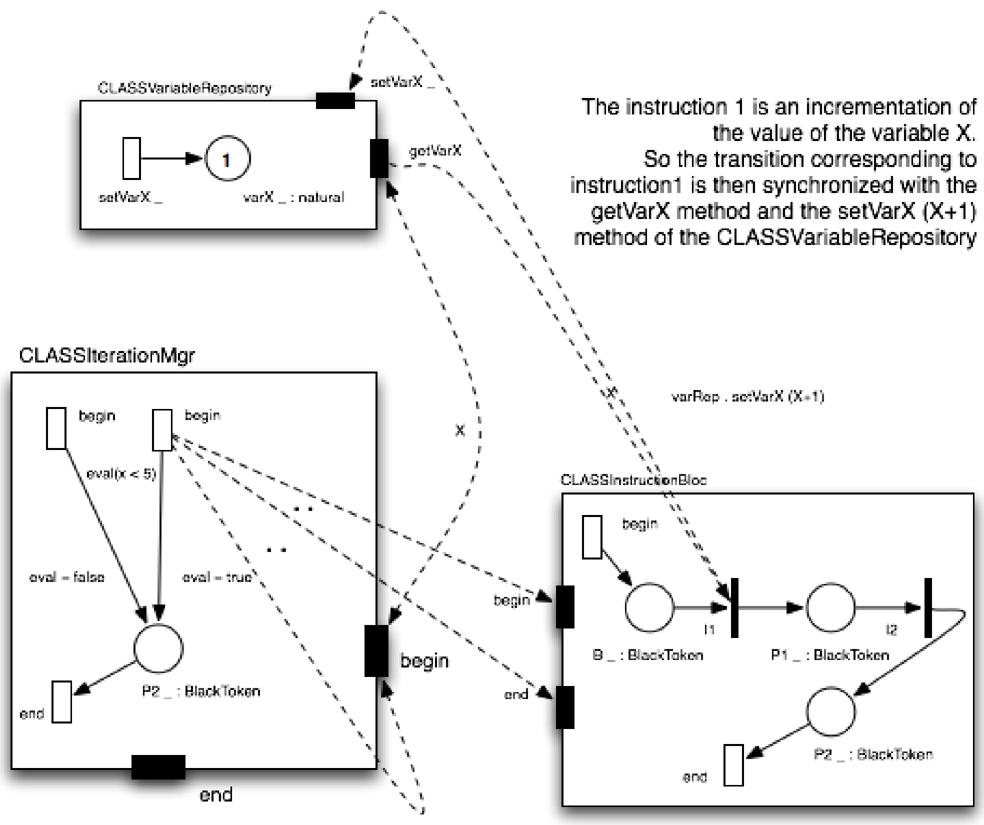


Figure 4.13. Graphic view of the COOPN Iteration Manager Class and interactions with other objects. Example: while (x smaller than 5) then (x++;)

4.3 Domain Concepts Transformations

This section presents the implementation details about the meta model instance to COOPN Specification transformation of the three domain concepts. The transformations are coded in Java language.

4.3.1 Assignment Domain Concept Transformation

As input we have an instance of the AssignmentDC Meta Model.

We are going to create then:

- a COOPN Class that will be the Variable repository;

- a COOPN Context that will contain a doAssignment method that will execute in sequence all the assignments.

First we create the CLASSVariableRepository. Then, for each assignment contained in the input instance we will create:

- a Place per variable which type will be the type of the Variable in the instance of AssignmentDC meta model;
- a getter and setter methods for accessing and manipulating the place content.

Note that for each type of variable, we have two possible case. If the type is contained in CFC we then just need to add the Interface Use to the Class. But if it is not a CFC Type, we also need to create an ADT to this corresponding type (and create its sort and generators).

We can now create the COOPN Context. We need to create:

- the Interface Use (using the created CLASSVariableRepository, and Types of places);
- an Interface method called doAssignment;
- an Object of type CLASSVariableRepository;
- an Axiom for expressing the fact that when the event doAssignment occurs, we will then call in sequence the setter methods for each variable that need to be assigned.

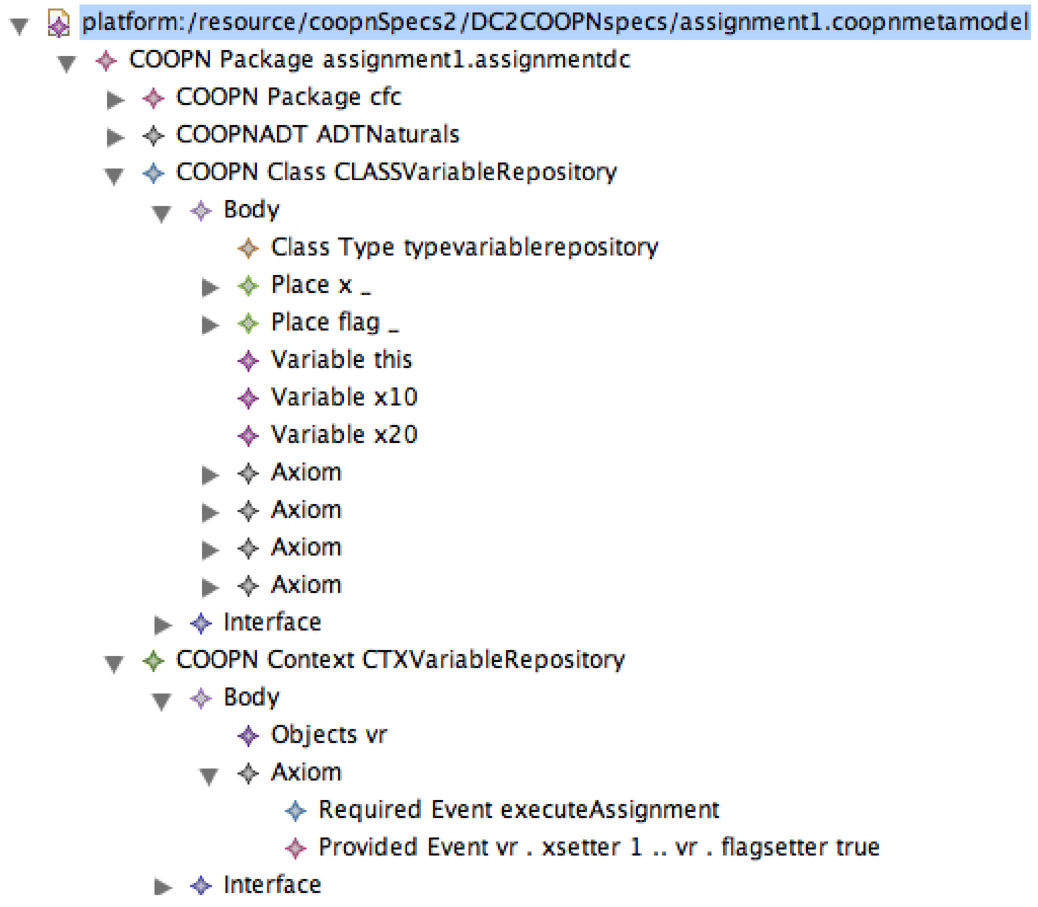
You can see in Fig. 4.14 an example of resulting generated COOPN specification.

4.3.2 Condition Domain Concept Transformation

As input we have an instance of the ConditionDC Meta Model.

We are going to create then for each Condition:

- a COOPN Class for Positive Instruction Bloc;
- a COOPN Class for Negative Instruction Bloc;
- a COOPN Class for Condition Manager;



```
▼ platform:/resource/coopnSpecs2/DC2COOPNspecs/assignment1.coopnmetamodel
  ▼ COOPN Package assignment1.assignmentdc
    ▶ COOPN Package cfc
    ▶ COOPNADT ADTNaturals
    ▼ COOPN Class CLASSVariableRepository
      ▼ Body
          Class Type typevariablerepository
        ▶ Place x _
        ▶ Place flag _
          Variable this
          Variable x10
          Variable x20
        ▶ Axiom
        ▶ Axiom
        ▶ Axiom
        ▶ Axiom
      ▶ Interface
    ▼ COOPN Context CTXVariableRepository
      ▼ Body
          Objects vr
        ▼ Axiom
            Required Event executeAssignment
            Provided Event vr . xsetter 1 .. vr . flagsetter true
      ▶ Interface
```

Figure 4.14. Generated COOPN Specification by the transformation from AssignmentDC Meta Model.

- a COOPN Context that will contain a doConditions method that will execute in sequence all the conditions.

The structure of the COOPN Classes for Instruction Blocs is the following:

- a Place of CFC type BlackTokens called B (for begin) and, if needed (not empty bloc of instructions), we will create places called Pi (with i from 0 to number of instructions of the bloc) of type BlackTokens;

- a Begin and a End methods;

- a set of Transitions called Ii (i from 0 to size of the Instruction set) - this set can be empty in case of empty bloc of instructions;
- a set of Axioms for expressing the fact that the Begin method adds a token to place B and synchronizes with Transitions (if there are some) and that the method End will consume a token in the last place;

It will be possible, thinking in terms of composition of these transformations, to synchronize the Transitions Axioms with a CLASSVariableRepository in order to realize an instruction that could be of type of Assignment.

Please note that this instruction bloc structure can be reused for iteration transformation presented in next subsection.

For the Condition Manager Class, we will have to create:

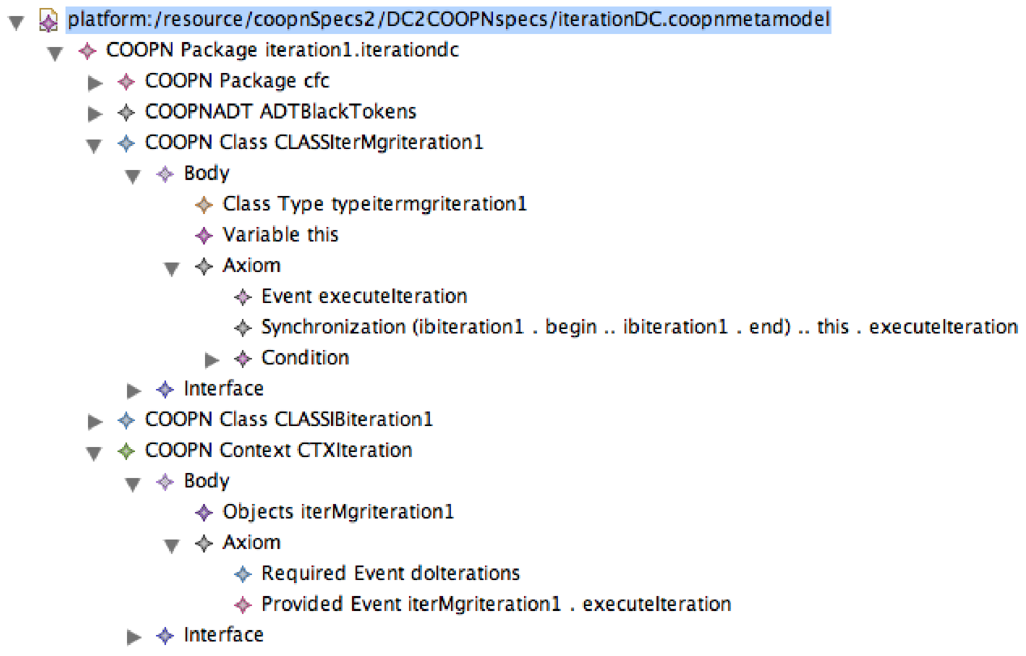
- an executeCondition method;
- two Objects of type of the Positive and Negative Instruction Bloc Classes presented before;
- a set of Axioms for expressing that if the result of the evaluation of the Boolean Expression is true we then call in sequence the begin and end of the Positive Instruction Bloc and if it is evaluated to false we then call the begin and end of the Negative Instruction Bloc;

The result of the Boolean Expression evaluation should be implemented in the transformation. As if for example we have a smaller than 5, we will then have to check in the content of the Place of the CLASSVariableRepository.

We can now create the COOPN Context. We need to create:

- the Interface Use;
- an Interface method called doConditions;
- an Object of type Condition Manager for each Condition;
- an Axiom for expressing the fact that when the event doConditions occurs, we will then call in sequence the executeCondition of each Condition Manager object.

You can see in Fig. 4.15 an example of resulting generated COOPN specification.



```
▼ platform:/resource/coopnSpecs2/DC2COOPNspecs/iterationDC.coopnmetamodel
  ▼ COOPN Package iteration1.iterationdc
    ▶ COOPN Package cfc
    ▶ COOPNADT ADTBlackTokens
    ▼ COOPN Class CLASSIterMgriteration1
      ▼ Body
          Class Type typeitermgriteration1
          Variable this
        ▼ Axiom
            Event executeIteration
            Synchronization (ibiteration1 . begin .. ibiteration1 . end) .. this . executeIteration
          ▶ Condition
      ▶ Interface
    ▶ COOPN Class CLASSIBiteration1
    ▼ COOPN Context CTXIteration
      ▼ Body
          Objects iterMgriteration1
        ▼ Axiom
            Required Event doIterations
            Provided Event iterMgriteration1 . executeIteration
      ▶ Interface
```

Figure 4.15. Generated COOPN Specification by the transformation from ConditionDC Meta Model.

4.3.3 Iteration Domain Concept Transformation

As input we have an instance of the IterationDC Meta Model.

We are going to create then for each Iteration:

- a COOPN Class for Instruction Bloc;
- a COOPN Class for Iteration Manager;
- a COOPN Context that will contain a doIterations method that will execute in sequence all the iterations.

The structure of the COOPN Classes for Instruction Blocs is the same as presented in previous section about Condition transformation.

For the Iteration Manager Class, we will have to create:

- an executeIteration method;
- an Object of type of Instruction Bloc Classe;

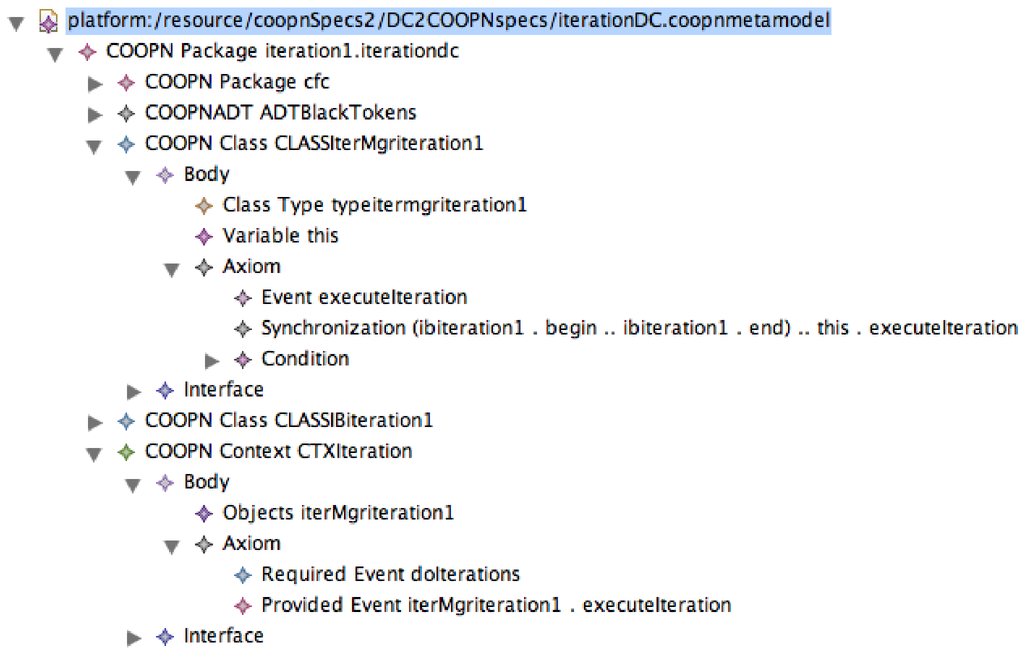
- a set of Axioms for expressing that if the result of the evaluation of the Boolean Expression is true we then call in sequence the begin and end of the Instruction Bloc and call recursively the executeIteration method;

The result of the Boolean Expression evaluation should be implemented in the transformation. As if for example we have a smaller than 5, we will then have to check in the content of the Place of the CLASSVariableRepository. For ending the recursion, we also need that the variable of the Boolean Expression should be changed as instruction in the Instruction Bloc.

We can now create the COOPN Context. We need to create:

- the Interface Use;
- an Interface method called doIterations;
- an Object of type Iteration Manager for each iteration;
- an Axiom for expressing the fact that when the event doIterations occurs, we will then call in sequence the executeIteration of each Iteration Manager object.

You can see in Fig. 4.16 an example of resulting generated COOPN specification.



```
▼ platform:/resource/coopnSpecs2/DC2COOPNspecs/iterationDC.coopnmetamodel
  ▼ COOPN Package iteration1.iterationdc
    ▶ COOPN Package cfc
    ▶ COOPNADT ADTBlackTokens
    ▼ COOPN Class CLASSIterMgriteration1
      ▼ Body
          Class Type typeitermgriteration1
          Variable this
        ▼ Axiom
            Event executeIteration
            Synchronization (ibiteration1 . begin .. ibiteration1 . end) .. this . executeIteration
          ▶ Condition
      ▶ Interface
    ▶ COOPN Class CLASSIBiteration1
    ▼ COOPN Context CTXIteration
      ▼ Body
          Objects iterMgriteration1
        ▼ Axiom
            Required Event doIterations
            Provided Event iterMgriteration1 . executeIteration
      ▶ Interface
```

Figure 4.16. Generated COOPN Specification by the transformation from IterationDC Meta Model.

Chapter 5

Meta Model Composition Tool

In this chapter, we will first start with a small introduction to ATL [14] language. ATL is a language for model transformation developed by the University of Nantes.

After that, you will find the problem approaches chosen for the different parts of the meta model composition tool.

5.1 ATL Model Transformation Language

ATL [14] is the ATLAS INRIA AND LINA research group answer to the OMG MOF [12] / Query/View/Transformation (QVT) [18] Request For Proposal (RFP). It is a model transformation language specified both as a metamodel and as a textual concrete syntax. It is a hybrid of declarative and imperative. The preferred style of transformation writing is declarative, which means simple mappings can be expressed simply. However, imperative constructs are provided so that some mappings too complex to be declaratively handled can still be specified. Once complex mappings patterns are identified, declarative constructs can be added to ATL in order to simplify transformation writing.

An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. The work on ATL is a collaboration between the University of Nantes and Institut National de Recherche en Informatique et en Automatique (France) (INRIA) and initially with TNI company. ATL has been chosen as the model transformation technology for the ModelWare IST European project in collaboration with SINTEF (Norway). It is currently being used by several research groups working in different domains and also for teaching.

5.1.1 ATL Execution Engine Architecture

A model-transformation-oriented virtual machine has been defined and implemented to provide execution support for ATL while maintaining a certain level of flexibility. As a matter of fact, ATL becomes executable simply because a specific transformation from its metamodel to the virtual machine bytecode exists. Extending ATL is therefore mainly a matter of specifying the new language features execution semantics in terms of simple instructions: basic actions on models (elements creations and properties assignments).

This flexibility is important for two main reasons: ATL will need to be aligned with the QVT standard when it is adopted in 2005 and, as a research project, it can this way easily benefit from newly defined features.

5.1.2 Available Developing Tools For ATL

An Integrated Development Environment (IDE) has been developed for ATL on top of Eclipse: ATL Development Tools (ADT). It uses EMF to handle models: to serialize and deserialize them, to navigate and to modify them. A specific code editor, including syntax highlighting and an outline view of the program, is implemented as a convenience.

This IDE also includes a specific ATL extension of the Eclipse debugging framework enabling source-level debugging of transformation programs. Single step, step over and breakpoints support makes it possible for the developer to precisely control the execution of the transformation program being written. When the execution is suspended, it is possible to navigate into source and target models from the current context as well as into user-defined variables. ADT is about to be released as part of the Eclipse Generic Mapping Tools (GMT) project under the Eclipse Public License (EPL).

5.2 A Meta Model For Meta Model Composition

We defined a meta model for meta model composition. This meta model describes the differents operators and elements needed for performing a meta model composition.

With this meta model, we can then generate in Eclipse the classes for having an editor. We can create instances of compositions with this editor or modify them. Those instances are going to be the input of our tool for automatical meta model composition.

The meta model for meta model composition can be seen on Fig. 5.1.

A composition is composed of at least one meta model, one operator and composition parameters (for knowing if we need to compose the meta model instances and/or transformations). The definition of the operators can be found in a following subsection.

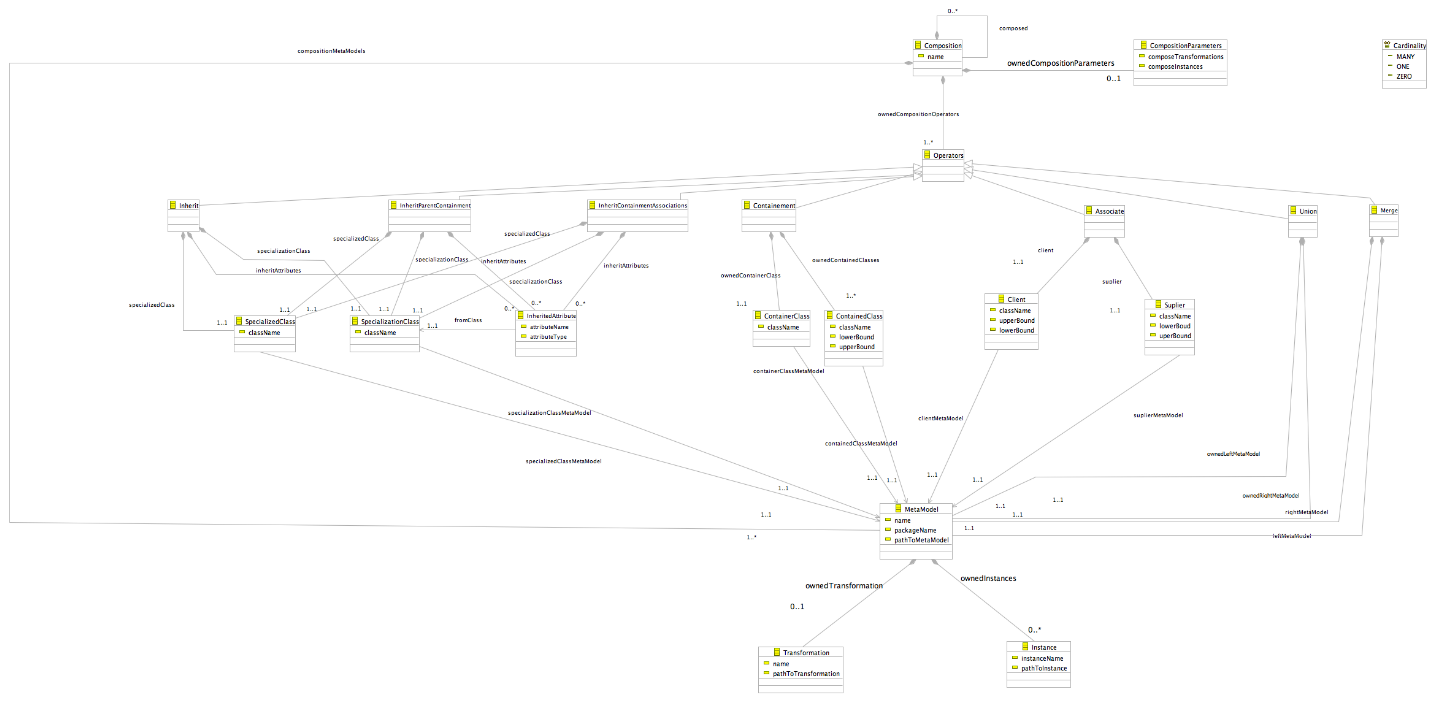


Figure 5.1. Meta model for meta model composition

5.3 Description Of Meta Model Operators

This section describes the semantic of the operators defined for meta model composition. Its objective is to help understanding the meta model defined for the meta model composition.

Here is a list of operators defined in a first attempt:

- Inherit;
- ImplementationInherit;
- InterfaceInherit;
- Containment;
- Association;
- Union;
- Merge;
- Difference;

But finally, we decided to implement only this list of operators for the meta model composition tool:

- Inherit;
- Containment;
- Association;
- Union;
- Merge;

In the following sections, you will find descriptions for all the operators defined. But finally we decided only to implement the previous list.

See the Fig. 5.2 of the meta model part describing the three Inherits operators.

First, we should have some words about the fact that we have three different Inherit operators.

We can find in an article Metamodel Composition in the Generic Modeling Environment [19] the following text:

'The Generic Modeling Environment (GME) metamodeling language is based on UML class diagrams. However, to support metamodel composition, some new operators are necessary.

The equivalence operator is used to represent the union of two UML class objects. The two classes cease to be separate entities, but form a single class instead. Thus, the union includes all attributes, compositions and associations of each individual class. Equivalence can be thought of as defining the join points or composition points of two or more source metamodels.

New operators were also introduced to provide finer control over inheritance.

When the new class needs to be able to play the role of the base class, but its internals need not be inherited, we use interface inheritance. In this case, all associations and those compositions where the base class plays the role of the contained object are inherited.

On the other hand, when only the internals of a class are needed by a subclass, we use implementation inheritance. In this case, all the attributes and those compositions where the base class plays the role of the container are inherited.

Notice that the union of these two new inheritance operators is the regular UML inheritance as illustrated in the figure below.'

We can find in an article Advanced Topics in Database Research [20] the following text:

'Implementation inheritance propagates all of the parents attributes, but only the containment association - where the parent functions as the container - to the child type. No other associations are inherited in this case.

Interface inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited.

Note: the union of the two special inheritance operators gives the common inheritance.'

Lets now describe each one of the mentioned operators.

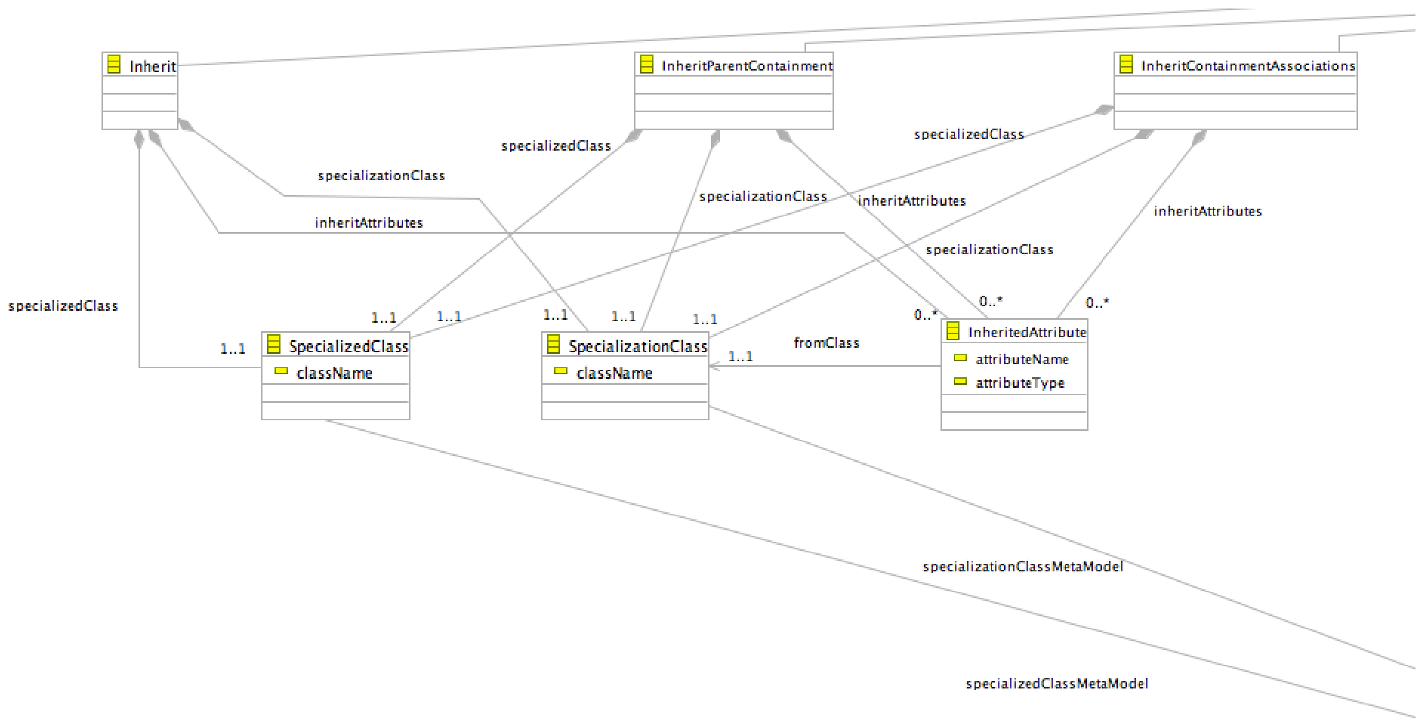


Figure 5.2. Inherit Operators in meta model for meta model composition

5.3.1 Inherit operator

The Inherit operator takes as parameters:

- A specialization class from one meta model;
- A specialized class from the other meta model;

With this operator, we are going to create the specialization relation between the two classes and inheriting the attributes of the specialization class to the specialized class. It is the inheritance relation as defined by UML .

5.3.2 ImplementationInherit operator

The ImplementationInherit operator takes as parameters:

- A specialization class from one meta model;
- A specialized class from the other meta model;

With this operator, the children inherit all of the parent attributes, but only the containment associations where the parent acts as the container.

Example: - Assuming that we have applied the ImplementationInherit (notation triangle with a white circle as defined by Meta GME) operator to the class B1 from a Meta model and to the class X1 from the other Meta model.

See the Fig. 5.3.

X1 inherits:

- The age attribute from B1;
- The association allowing object of type C1 to be contained in objects of type B1 (X1 can contain C1 objects);
- The association allowing objects B1 to be contained in B1 (X1 can contain B1 objects but not X1 objects);
- The D1 objects can contain B1 objects but not X1 objects;

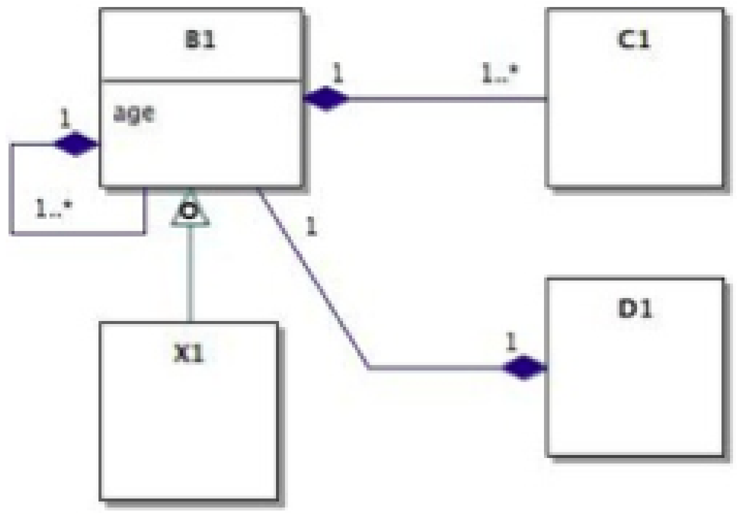


Figure 5.3. ImplementationInherit example

5.3.3 InterfaceInherit operator

The InterfaceInherit operator takes as parameters:

- A specialization class from one meta model;
- A specialized class from the other meta model;

This operator means that the inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited.

Example: - Assuming that we have applied the InterfaceInherit operator to the class B2 from a Meta model and to the class X2 from the other Meta model.

See the Fig. 5.4.

X2 inherits:

- The X2 objects can be contained in objects of type D2 and B2;
- No objects can be contained in X2 (not even other X2 objects);
- The age attribute is not inherited by X2;

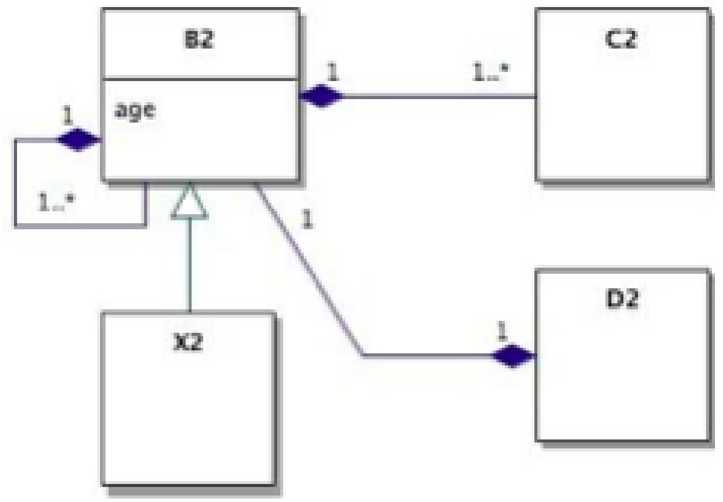


Figure 5.4. InterfaceInherit example

5.3.4 Containment

The Containment operator takes as parameters:

- A ContainerClass from one meta model;
- A list of ContainedClass (with association end lower and upper bounds) from the other Meta model;

This operator simply creates one or various containment association between a class of the left Meta model and a list of classes of the right Meta model.

5.3.5 Association

The Association operator takes as parameters:

- A Client class in one meta model with upper and lower association end bounds;;
- A Suplier class from the other Meta model with upper and lower association ends bounds;

This operator creates an association relation between one class of the left Meta model and one class of the right Meta model.

5.3.6 Union

The Union operator takes as parameters:

- 2 Meta models;

This operator will consider all elements of the two Meta models as distinct elements and makes the union between them.

Example: - Assuming that we are going to apply the Union operator to those two meta models.

See the Fig. 5.5.

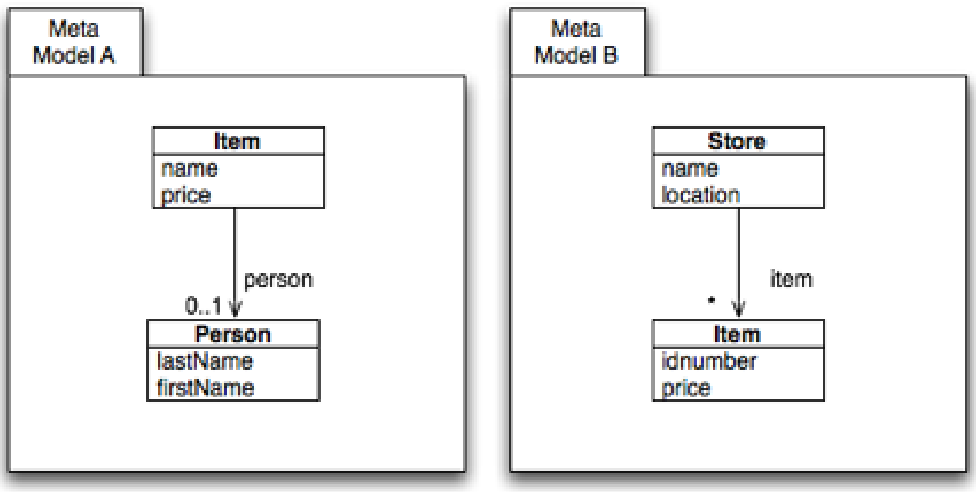


Figure 5.5. Union example 1

The question is what is going to happen with the Class called Item, which is present in both meta models. For the union, we consider that those two classes are distinct elements, so in the result composed meta model, we are going to rename the class of the second meta model. Here is a picture describing the result of the composition.

See the Fig. 5.6.

An important question that appears now is how we should rename the class from the second meta model? We should be careful because we do not want to give a name of a class that could already exist. In the picture for the example, the new name is generated concatenating a RMM (stands for

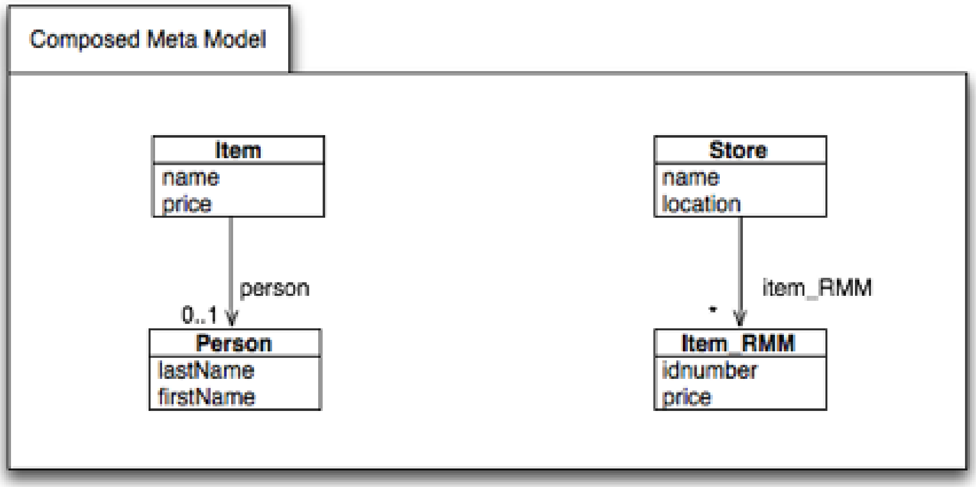


Figure 5.6. Union example 2

Right Meta Model) to the old name. This renaming convention should avoid any problem.

We also need to rename all the associations names of the class affected by the renaming. We can rename those associations with the same renaming convention of adding a RMM to the old association name. This allows to avoid problems when dealing with elements of meta model as there will be elements for each association.

5.3.7 Merge

The Merge operator is a more refined Union. That means that, instead of considering each element of the two Meta models as distinct, we want to deal with the case of having elements considered as equal and that we can merge.

How can we decide that two elements of two distinct Meta models are equal? And at which level are we talking about: class elements? Attribute elements? Association elements?

Lets define what we understand by equality for each of these elements.

Class element equality: We define that two classes having the same class name are equal.

Attribute element equality: We define that two attributes of two classes with the same class name are equal if and only if the attribute name and the attribute type are equal.

Association element equality: We define that an association that links two classes that are equal and that have the same source and target class names, the same association name and the same association ends (multiplicity) are equal.

So, now that we have defined the equality, we can define the Merge operator. This operator makes an union with all the distinct elements but, if a class from the left meta model is considered equal (according to our definition of equality) to a class of the right meta model, we are going to merge this class. This means that where there were two elements, in the result meta model there will be only one.

Let see with some examples what cases we can have.

Example 1: - We assume that we are using the Merge operator between those two meta models.

See the Fig. 5.7.

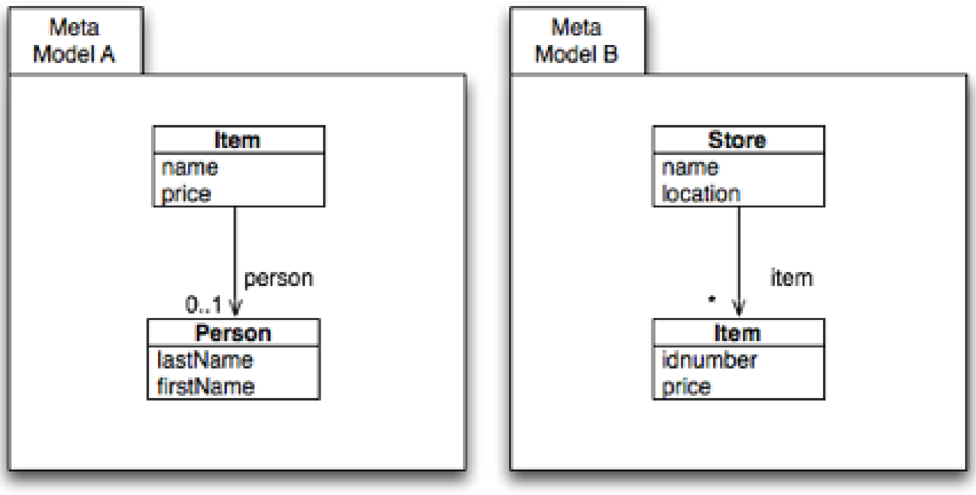


Figure 5.7. Merge example

We have here two classes that are equal because they have the same class name (equality definition). So we know that we are going to merge those classes. But, we see that we are confronted to some issues: - Should we do an union of the attributes; - Should we do a merge of the attributes that are equal (with the defined attribute equality); - In the case we had also a class Person with the same association to the Item class in the meta model B, should we merged it also? - In the case we had also a class Person but with an association that was not equal, should we still merge it? Should we keep two different associations?

Lets take each case and see what the result of the operator is depending on the case.

In this case, we assume that Item classes are equal and that Price attributes are equal (same attribute name and same attribute type). So we get a merged Item class with 3 attributes: name, idnumber and price. In this case we merged the two classes that were equal (Item) and the two attributes that were equal (price).

See the Fig. 5.8.

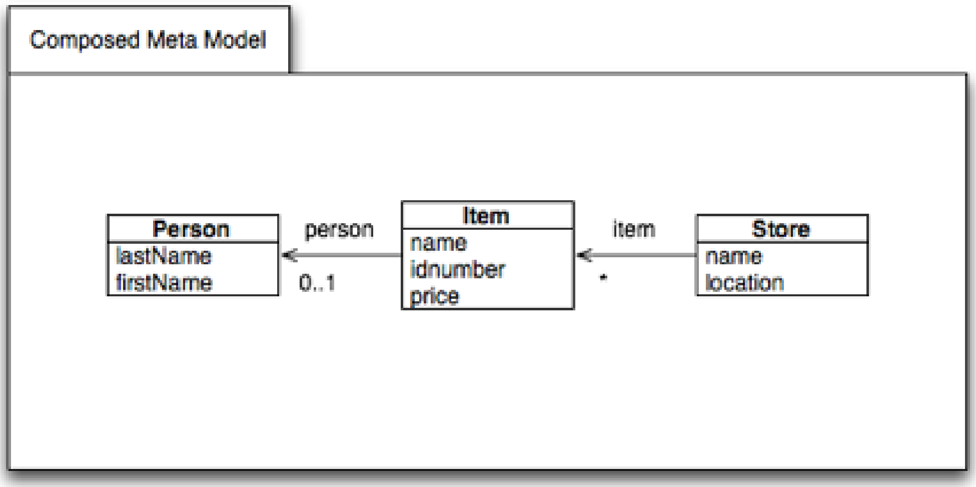


Figure 5.8. Merge example

The natural question we can have at this point is what would have happened if the price attributes were not equal having for example the same name but different types. Lets assume for example that the price attribute type in meta model A is of type String and that the type of the price attribute in the meta model B is Integer as it can be seen in the next picture.

See the Fig. 5.9.

So in this case, we decided that the two attribute of the merged class

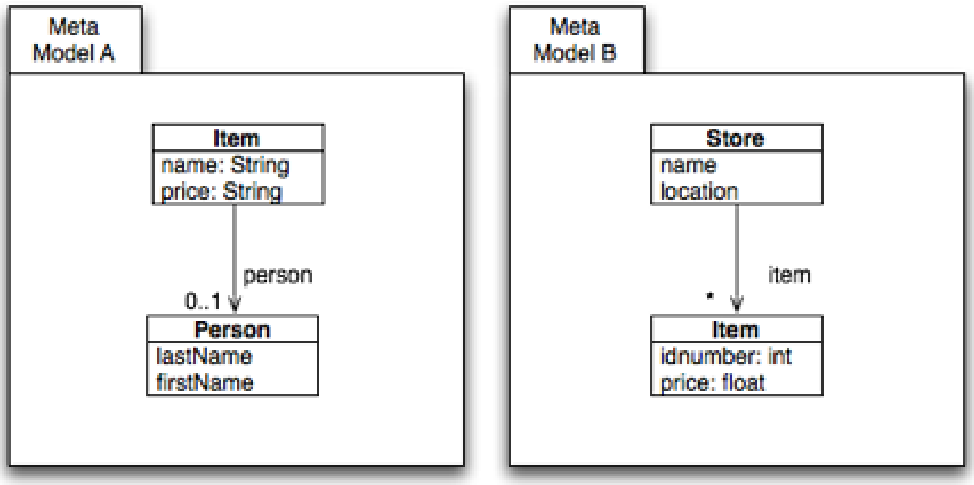


Figure 5.9. Merge example

that are not equal but have the same name should be kept. But, we are going to rename the attribute from the right meta model with the renaming convention defined for the Union operator. The resulting meta model would look like the following picture.

See the Fig. 5.10.

Now, we should see the case where we have an association merge. Look at the picture below.

See the Fig. 5.11.

In this case, we clearly see that we are going to merge the classes Item and Person, all the attributes of these classes that are equal (same name and same type) and also the association called person which is equal because it has same name, same target and source names and same association ends multiplicity.

In this case the result can be seen on next picture, which is the same as for the precedent case.

See the Fig. 5.12.

But, this example makes us think on a new possible issue. What would have happened is the new class in the right meta model called Person had a different name? Lets study this case in the following picture.

See the Fig. 5.13.

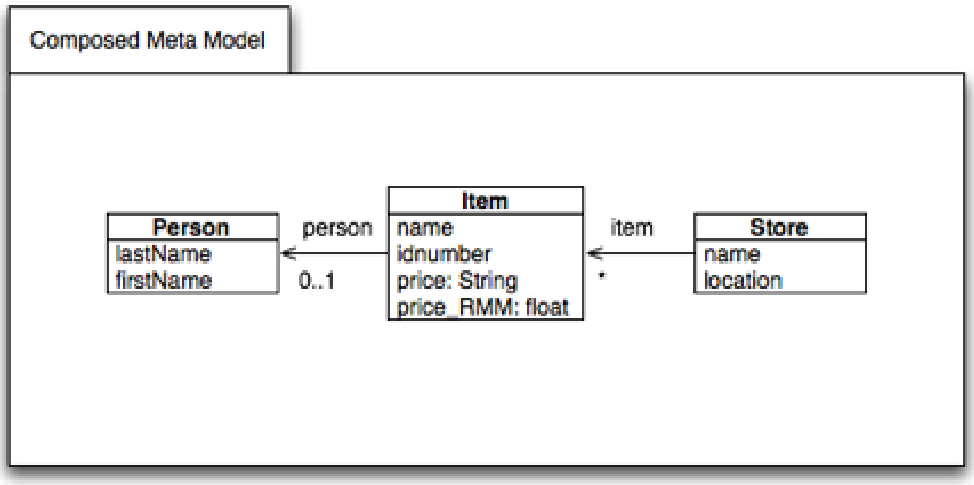


Figure 5.10. Merge example

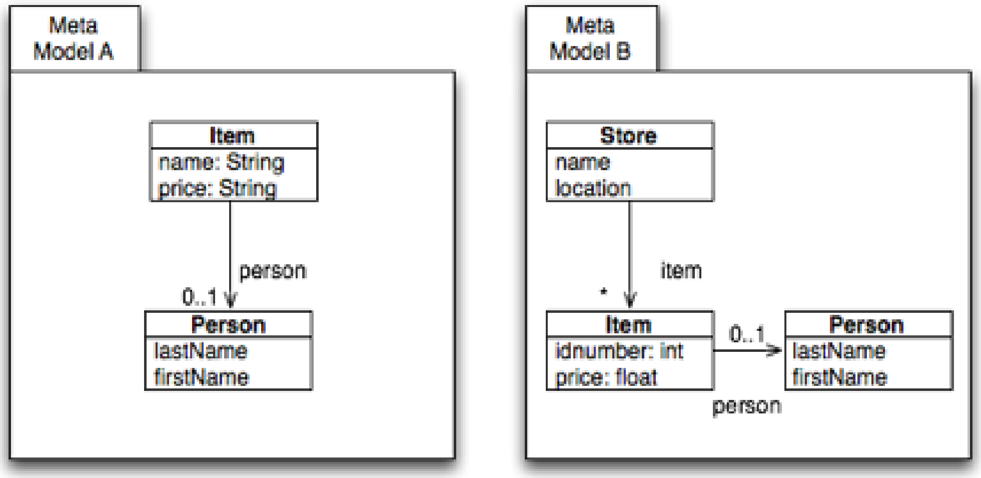


Figure 5.11. Merge example

Here we have a class Client with an association called person. We should manage the fact that when we are going to merge those two meta models we are going to have in the resultant composed meta model to deal with the problem of having two different association with the same name. In this

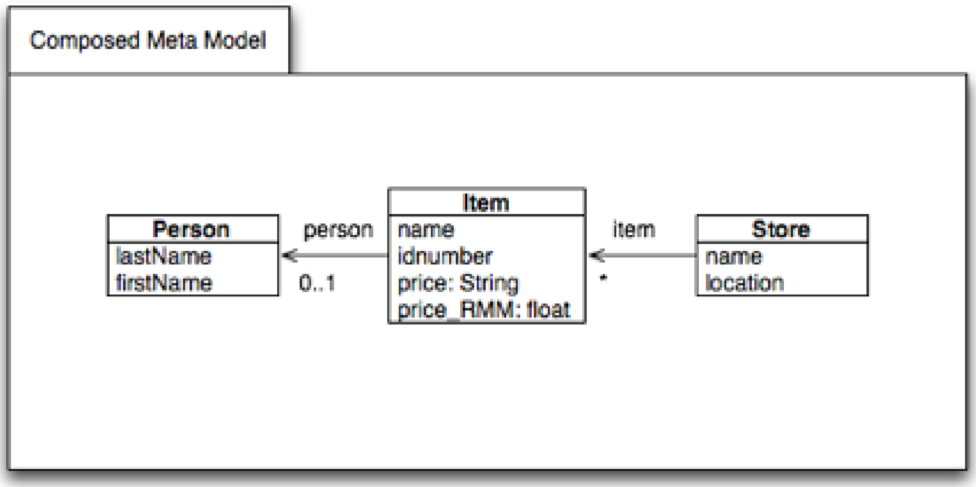


Figure 5.12. Merge example

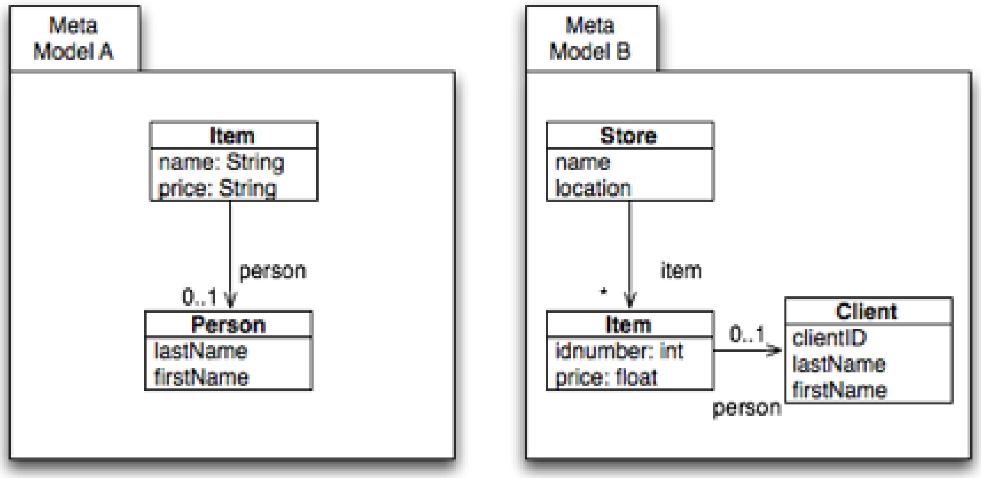


Figure 5.13. Merge example

case, we are going to rename the association name from the right meta model, which will give us the result as presented in the picture below.

See the Fig. 5.14.

There is still one case we have to look at. In the next picture we have

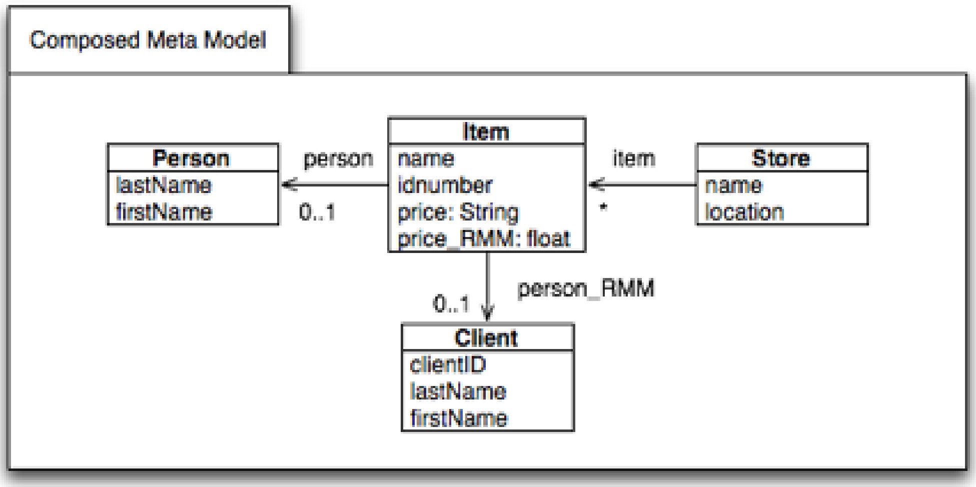


Figure 5.14. Merge example

the case where we are going to merge the classes Persons and Items but the association names are not the same.

See the Fig. 5.15.

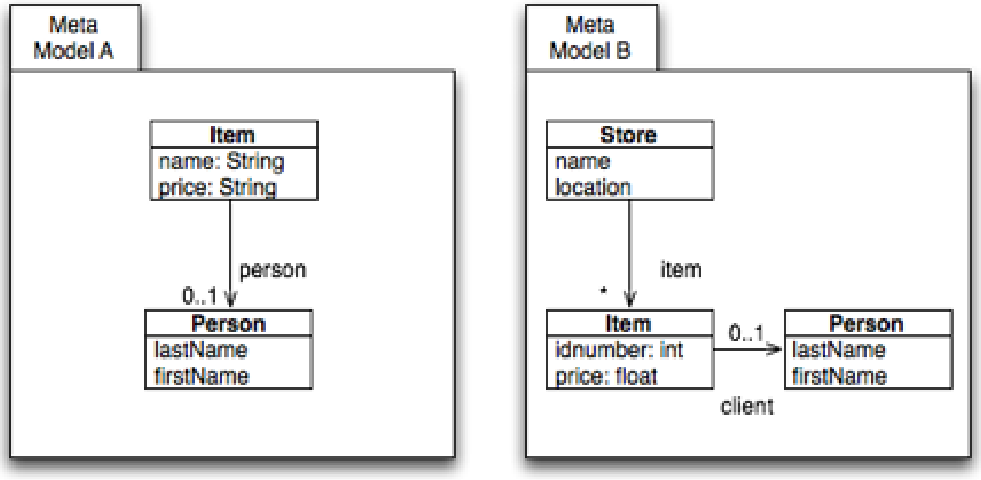


Figure 5.15. Merge example

We can see that the associations are not equal because their names are not the same (person in left meta model, client in the right meta model). So, as result we are going to merge the classes, some of the attributes as described before, but we cannot merge the association. So the result will look as presented in this picture.

See the Fig. 5.16.

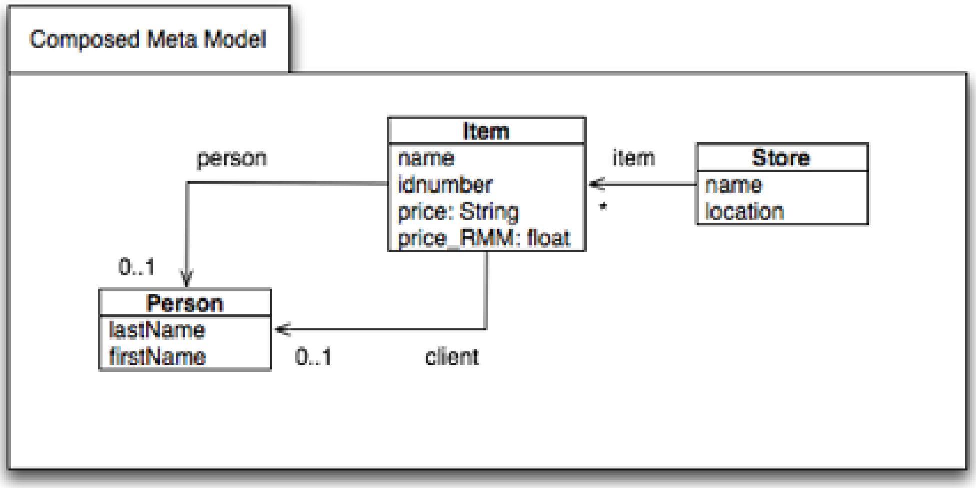


Figure 5.16. Merge example

5.3.8 Difference

The Difference operator allows producing a new meta model with all the elements that are considered as different in the 2 meta models. Two class elements are different if and only if they are not equal (considering our equality definition). This means that all the classes that have the same name will not be present in the resulting composed meta model.

5.4 Transformation Of Meta Models With Operators

In this section we will see how the implementation of the operators for meta model composition have been implemented. The idea is to develop a tool that will take as input two meta models (which conforms to the Ecore meta meta model) and an operator and will produce as output a resulting meta model (which will conform to Ecore too) that will be the result of the composition of the two input meta models using the chosen operator.

This tool is developed in Java but uses the ATL language and the ATL Tool API for executing some predefined ATL transformation templates. In the following subsection you will find the main idea. Then each ATL transformation template will be presented. In the last subsection, some information and tips will be given about the Java tool implementation.

5.4.1 Meta Model Composition Tool - Main Idea

The tool for meta model composition takes as input two meta models (conforming to Ecore) and an operator name and produces as output the composed meta model (conforming to Ecore). The Fig. 5.17 presents the flow chart of this tool and helps to understand what are the main processes of the tool.

The user indicates the tool two meta models paths (the MM abbreviation in LeftMM and RightMM stands for Meta Model) and the operator he wants to use for the meta model composition.

The tool will then select the corresponding ATL transformation template depending on the given composition operator. An ATL transformation template has been defined for each operator. Those ATL transformations are fully presented in the following subsection.

Once the good ATL transformation template has been selected and loaded, the tool will adapt it and generate an ATL file that will allow the ATL transformation of the meta models to the composed meta model.

Now, using the ATLCompiler class defined by ATL developpers, the tool compile the specific ATL transformation file. This produces an ASM file.

The idea is now to finally use the ATLLauncher class in order to launch the ATL transformation. This produce the composed meta model, which is the expected output.

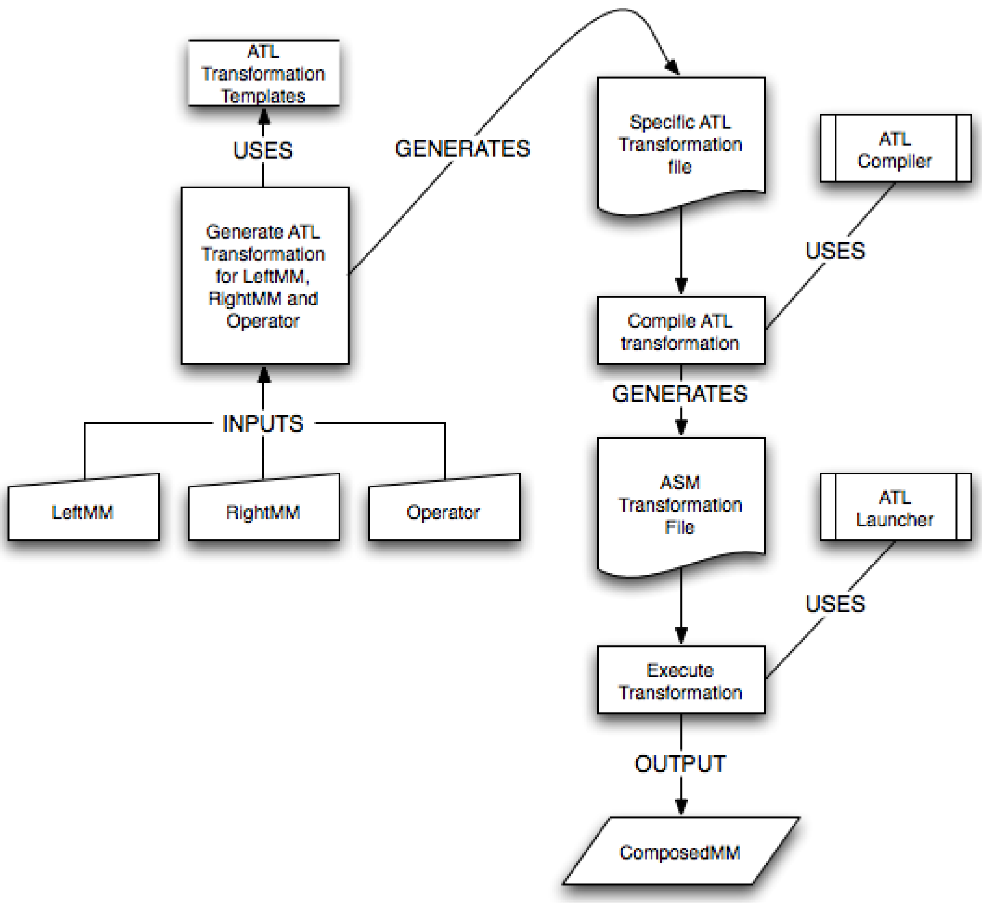

ATL Transformation Templates
USES
GENERATES
Specific ATL Transformation file
ATL Compiler
Generate ATL Transformation for LeftMM, RightMM and Operator
USES
Compile ATL transformation
GENERATES
INPUTS
LeftMM
RightMM
Operator
ASM Transformation File
ATL Launcher
USES
Execute Transformation
OUTPUT
ComposedMM

Figure 5.17. Flow Chart for Meta Model Composition Tool

The tool produces also a file that will be containing rules that will help us for the instances of meta model composition transformation. This instances transformation is performed in another tool presented further.

5.4.2 Some ATL Tips and Advices

As explained in previous subsections, ATL Transformation Templates have been defined for the meta model composition tool. Those templates have been defined using examples and those examples are presented in this subsection. For each ATL transformation, we take as input the two meta models (conforming to Ecore meta meta model) we produce as output the result of the composition with the operator which is a new meta model (conforming to Ecore too). Let's describe those ATL transformation with examples in the following sections.

Each operator has already been defined in details in previous chapters. Before we get into explaining each transformation, some general ATL tips and advices are given in following chapters.

ATL Transformation - Difficulties and Solutions

Here are some tips about how to use the ATL Eclipse Plugin. This information is fruit of my experience while learning and trying to get those ATL transformations working. I hope this will help people interested in trying meta model transformation with ATL.

1 - How to launch ATL transformation with multiple source patterns?

For meta model composition with ATL, we have two meta models as inputs and that produces a composed meta model as output. In order to be able in ATL to do transformation with multiple inputs models (in our case the left and right meta models) you should add the following line at the top of your transformation definition:

– @atlcompiler atl2006

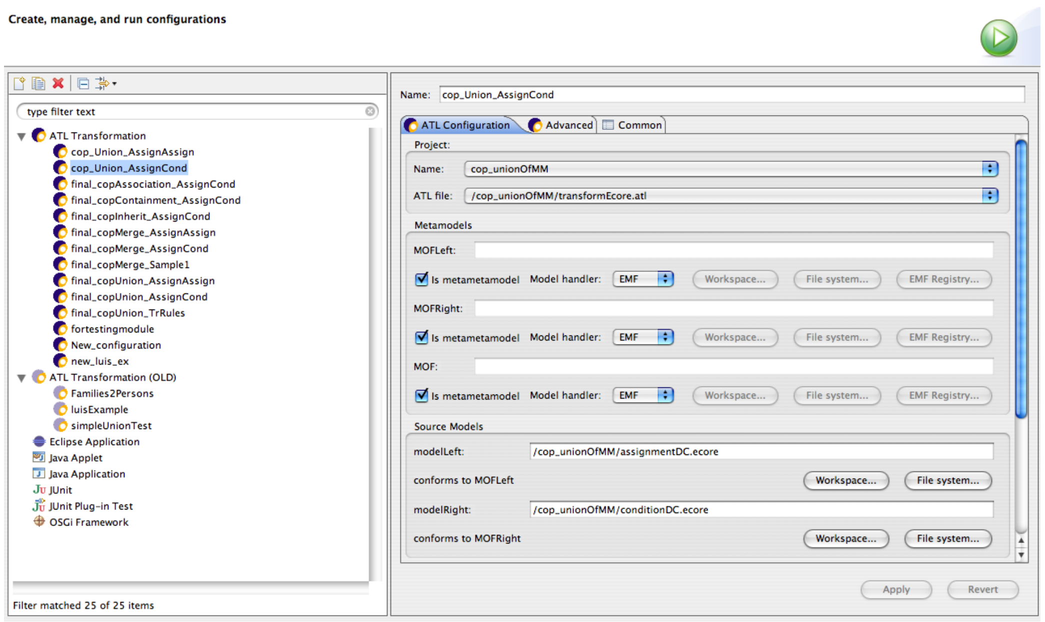
Thanks to Luis Pedro and Adil Anwar from the ATL newsgroup [21] for this tip.

2 - How to launch ATL Ecore to Ecore transformation

ATL transformation is widely used for model transformation. In this case, we define the input and output models and the meta models to which they conforms to. But, for an Ecore to Ecore transformation, as for meta model composition, the input and output models are in fact meta models and they conform to the Ecore meta meta model.

To launch an Ecore to Ecore ATL transformation, don't forget to check the "is meta meta model" option in the Run Configuration of the ATL Eclipse plugin. If you don't do that, you will get an error saying that allInstances operation could not be found.

See Fig. 5.18 to see an example of Ecore to Ecore transformation Run Configuration.


Create, manage, and run configurations

type filter text

- ATL Transformation
  - cop_Union_AssignAssign
  - cop_Union_AssignCond
  - final_copAssociation_AssignCond
  - final_copContainment_AssignCond
  - final_copInherit_AssignCond
  - final_copMerge_AssignAssign
  - final_copMerge_AssignCond
  - final_copMerge_Sample1
  - final_copUnion_AssignAssign
  - final_copUnion_AssignCond
  - final_copUnion_TrRules
  - fortestingmodule
  - New_configuration
  - new_luis_ex
- ATL Transformation (OLD)
  - Families2Persons
  - luisExample
  - simpleUnionTest
- Eclipse Application
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- OSGi Framework

Filter matched 25 of 25 items

Name: cop_Union_AssignCond

ATL Configuration | Advanced | Common

Project:

Name: cop_unionOfMM

ATL file: /cop_unionOfMM/transformEcore.atl

Metamodels

MOFLeft:

[✓] Is metametamodel Model handler: EMF Workspace... File system... EMF Registry...

MOFRight:

[✓] Is metametamodel Model handler: EMF Workspace... File system... EMF Registry...

MOF:

[✓] Is metametamodel Model handler: EMF Workspace... File system... EMF Registry...

Source Models

modelLeft: /cop_unionOfMM/assignmentDC.ecore

conforms to MOFLeft Workspace... File system...

modelRight: /cop_unionOfMM/conditionDC.ecore

conforms to MOFRight Workspace... File system...

Apply Revert

Figure 5.18. ATL Eclipse Plugin Run Configuration 'is metametamodel' option selected for Ecore to Ecore transformation - Meta model Union run configuration example.

3 - ATL lazy rules

In ATL transformation, we should define rules that will say what to produce and to do when a pattern is found in the source model or meta model. For example, in our Ecore meta model composition examples we defined a lazy rule for dealing with EClasses from the right meta model.

In this case, we are defining union of meta models. So we are going to define in the ATL transformation all the information of left meta model must be copied to resulting meta model. And for right meta model information, we can copy all the information but renaming the classes that have the same name than a class from left meta model.

This means that, before copying information from right meta model, we need to check if the name of the EClass is the same than one of the EClass contained in the left package. That was a problem because with simple rules I was not finding any way of getting this information.

So for doing that, we need to use a lazy rule because we need to get information from the left meta model EPackage.

The use of lazy rules for the union ATL transformation example can be seen in code listing in Appendix A.

For more information about lazy rules and other types of ATL rules, please consult the ATL User Manual [22].

For more details about the Union ATL Transformation, please see the corresponding paragraph.

4 - ATL entrypoint and endpoint

While defining my ATL meta model composition transformations, I have been faced to another problem. In the case of the union of meta models, when classes from left and right meta model were the same, we must rename the class name of the element coming from the right meta model.

Then a new problem appears. If you rename a class, you will have to adapt the associations (EReferences) of the renamed class. That seems to be easy said like that, but the problem is that for setting the good link for the EReference in the resulting meta model, some times we are trying to make the new ereference pointing to an EClass that has still not been created by the ATL Transformation!

In order to avoid that, we need to use an endpoint rule as can be seen in the following code listing.

We can now see in detail each one of the ATL transformation corresponding to the different composition operators.

Listing 5.1. Example of endpoint rule in ATL Transformation for Union operator

```
...

-- EXECUTION OF DELAYED ACTIONS

endpoint rule EndRule() {
  do {
    -- renaming and modifying eAttributes
    for(dta in thisModule.attributesFromResultMM) {
      dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eType.name))
            then
```

```
                (
                (MOF!EClass.allInstances()->any(e | e.name = dta.
                    eType.name))
                )
              else
                (
                (MOF!EClass.allInstances()->any(e | e.name = (dta.
                    eType.name + '_RMM')))
                )
            endif;
    }
    -- renaming and modifying eReferences
    for(dta in thisModule.referencesFromResultMM) {
      -- we can now rename the eReference name
      dta.name <- if(thisModule.referencesNamesFromLMM.includes(
          dta.name))
                then
                  (dta.name+'_RMM')
                else
                  (dta.name)
              endif;
      -- and modify the binding to the good EClass
      dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eReferenceType.name))
                then
                  (
                   (MOF!EClass.allInstances()->any(e | e.name =
                       dta.eReferenceType.name))
                  )
                else
                  (
                   (MOF!EClass.allInstances()->any(e | e.name = (
                       dta.eReferenceType.name + '_RMM')))
                  )
              endif;
    }
  }
}
```

The endpoint rule allows to execute some action at the end of the transformation process. In the example all the eAttributes and eReferences readapting are done in the endpoint rule as it could not be done before as we need to have in the result meta model all the resulting EClasses created.

For more information about entry and end points rules and other types of ATL rules, please consult the ATL User Manual [22].

5.4.3 ATL Transformations Templates

Here are described all the defined templates for each operator.

ATL Transformation for Inherit Operator

The Inherit operator produces as result meta model a meta model with the content of left and right meta models and will add an Inheritance relation between a class of left meta model and a class of right meta model.

By default, the class in the left meta model is the Super Class and the class of the right meta model will play the role of the Specialized Class.

The ATL transformation for that is pretty easy to understand. In the 'init' rule we just create the resulting EPackage, set its name and copy in it the eClassifiers of left and right meta model.

Then, we simply create in the resulting meta model all the EClasses, EAttributes and EReferences of both meta models with rules.

In order to create the inheritance link, we just need to add a eSuperType to the corresponding EClass from the right meta model. This is done in the 'CLASSRight' ATL rule. As you can see in the following code listing, an EClass from right meta model will produce the corresponding EClass for the result meta model, but, in the 'do' section, we defined that if the EClass name from right meta model is equal to the name of the class we want to be the specialization of the Inheritance relation, we will then add to it an eSuperType which will be the class of the resulting meta model coming from the left meta model indicated by the usage of the operator.

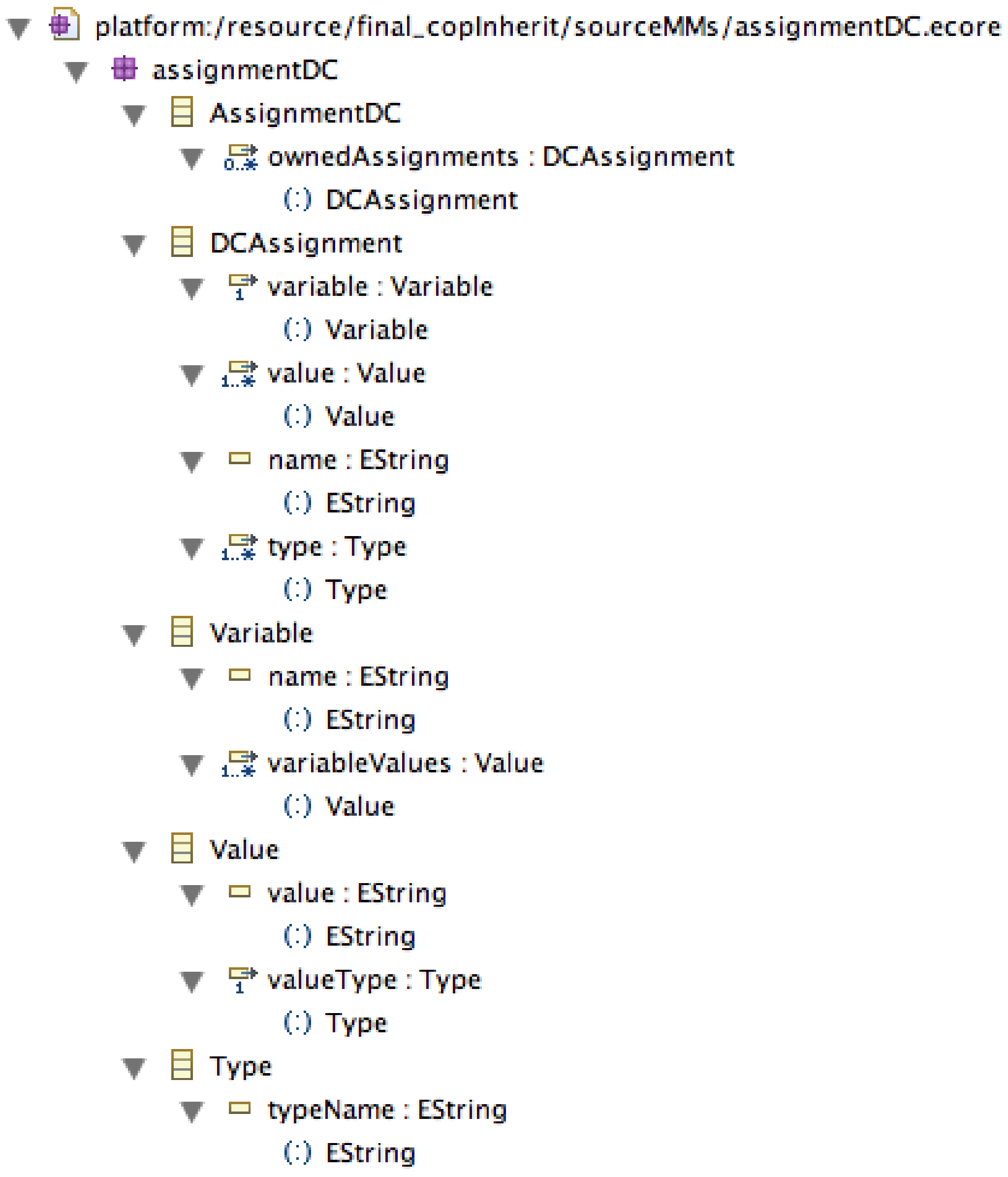
You can take a look to the Inherit Operator Template in Appendix A.

In this example, we worked with AssignmentDC and ConditionDC (DC stands for Domain Concept, and the meta models can be seen in the corresponding chapter) meta models. We supposed that a user wants to call the transformation for creating an Inheritance realtion between the AssignmentDC Class of the left meta model and the Instruction Class of the right meta model (the Instruction class being then the specialized class.

See AssignmentDC meta model in Fig. 5.19.

See ConditionDC meta model in Fig. 5.20.

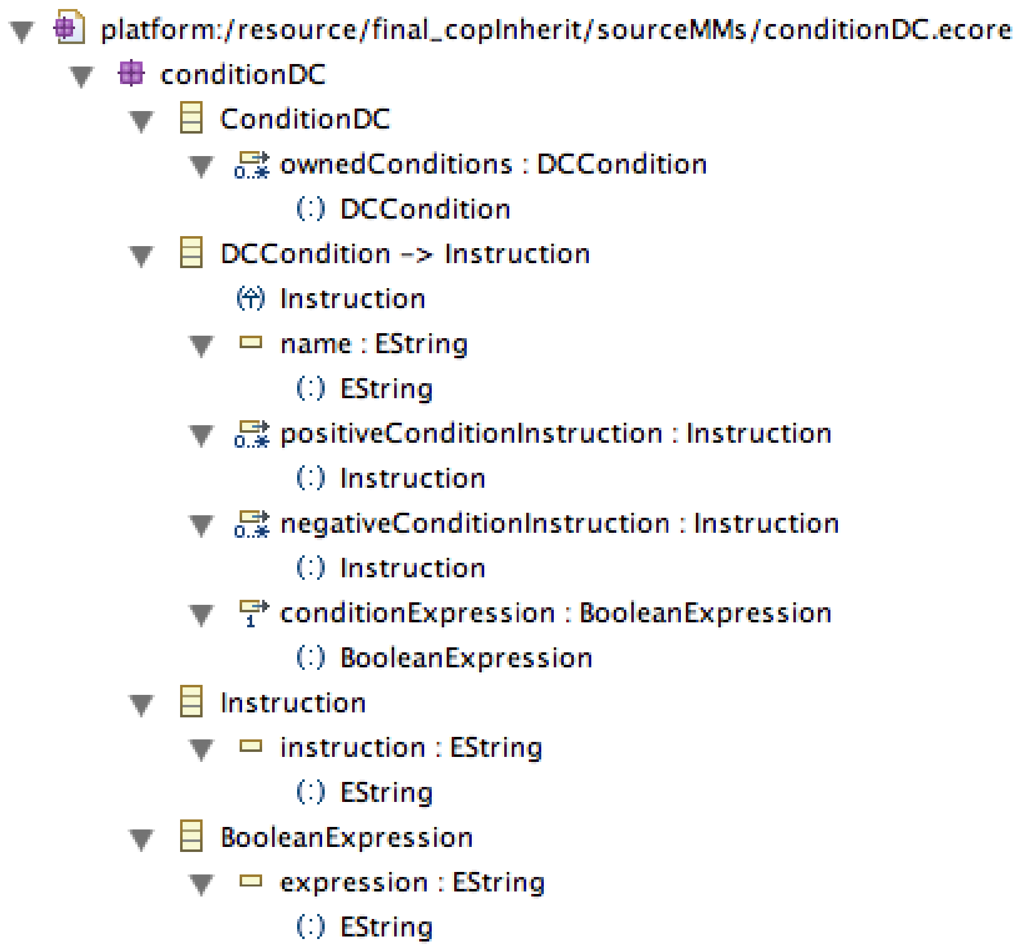
See AssignmentDCInheritConditionDC meta model in Fig. 5.21.



```
platform:/resource/final_copInherit/sourceMMs/assignmentDC.ecore
    assignmentDC
        AssignmentDC
            ownedAssignments : DCAssignment
                (:) DCAssignment
        DCAssignment
            variable : Variable
                (:) Variable
            value : Value
                (:) Value
            name : EString
                (:) EString
            type : Type
                (:) Type
        Variable
            name : EString
                (:) EString
            variableValues : Value
                (:) Value
        Value
            value : EString
                (:) EString
            valueType : Type
                (:) Type
        Type
            typeName : EString
                (:) EString
```

Figure 5.19. Assignment Domain Concept Ecore meta model

So the question now, is how can we get from this specific ATL transformation example a Template for our Meta model Composition tool? We will just remove the content of the 'do' part of the CLASSRight rule and the tool



```
platform:/resource/final_copInherit/sourceMMs/conditionDC.ecore
  conditionDC
    ConditionDC
      ownedConditions : DCCondition
        DCCondition
    DCCondition -> Instruction
      Instruction
      name : EString
        EString
      positiveConditionInstruction : Instruction
        Instruction
      negativeConditionInstruction : Instruction
        Instruction
      conditionExpression : BooleanExpression
        BooleanExpression
    Instruction
      instruction : EString
        EString
    BooleanExpression
      expression : EString
        EString
```

Figure 5.20. Condition Domain Concept Ecore meta model

will have to replace the specific to inputs meta models names of classes for creating the right inheritance relation requested by the user.

Please note that all the union problematic for this operator (same class names in left and right meta model for example) are not treated by us, it depends on ATL launcher and on the way they manage this type of problems.

For transformation of instance of meta models, the information we need to know is only the two class names. So the tool will generate a file with this useful information. For more details on instances transformation, please refer to the corresponding chapter.

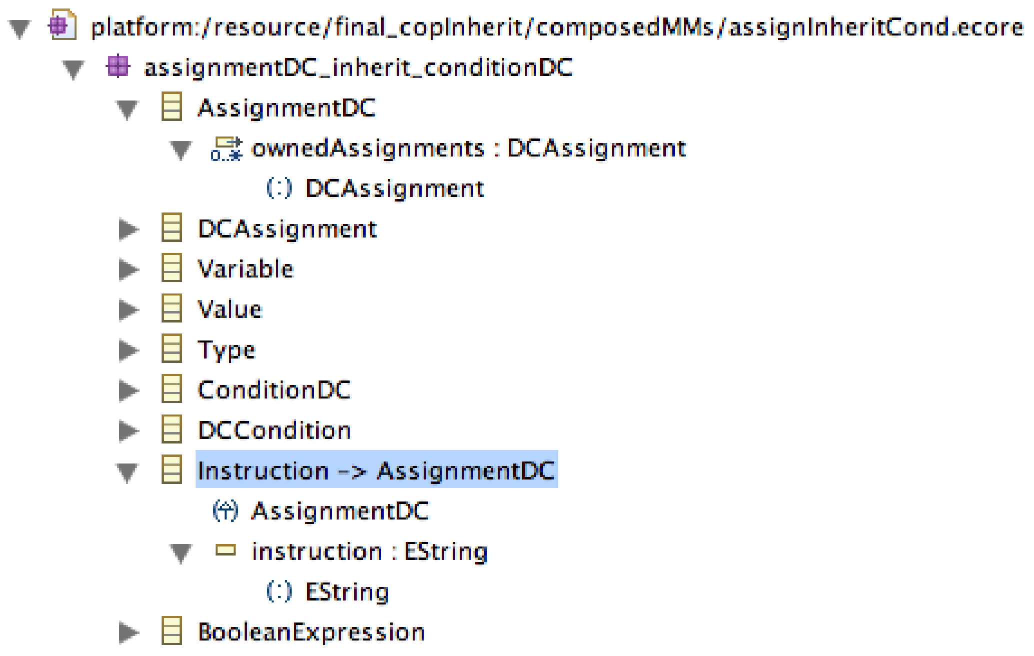

platform:/resource/final_copInherit/composedMMs/assignInheritCond.ecore
assignmentDC_inherit_conditionDC
AssignmentDC
ownedAssignments : DCAssignment
DCAssignment
DCAssignment
Variable
Value
Type
ConditionDC
DCCondition
Instruction -> AssignmentDC
AssignmentDC
instruction : EString
EString
BooleanExpression

Figure 5.21. AssignmentDCInheritConditionDC Ecore meta model, result of the ATL transformation

ATL Transformation for Containment Operator

The Containment operator produces as result meta model a meta model with the content of left and right meta models and will add a containement relation between a class of left meta model and one or more classes of right meta model.

The class from the left meta model is the Container and the classes of the right meta model will play the role of the Contained.

The ATL transformation for that is also easy to understand. In the 'init' rule we just create the resulting EPackage, set its name and copy in it the eClassifiers of left and right meta model.

Then, we simply create in the resulting meta model all the EClasses, EAttributes and EReferences of both meta models with ATL rules.

The ATL solution found for creating the containment relation is the following. Left EClasses are normally copied to resulting meta model. But for classes from the right meta model, we defined two ATL rules called 'CLASSRight1' and 'CLASSRight2'.

The first rule will take the classes from right meta model that are the contained classes. So for those classes, we create the resulting EClass plus an EReference for the containment relations. Why are we creating the eReference at this point? This is because when creating the containment eReference, we give it the name of the eClass, we set the lower and upper bounds, and the more importante we set the eType of the containment which is the freshly created EClass. We then also add the containment eReference to a list because we will then need to add those new eReference to the Container class from the left meta model. This is done in the delayed actions in the endpoint ATL rule.

The second rule ('CLASSRight2' ATL rule) will match all the classes from right meta model that are not contained elements and simply create the resulting EClass in the composed meta model.

You can take a look to the Containment Operator Template in Appendix A.

In this example, we worked with AssignmentDC and ConditionDC (DC stands for Domain Concept, and the meta models can be seen in the corresponding chapter) meta models. We supposed that a user wants to call the transformation for creating a Containment relation between the AssignmentDC Class of the left meta model (acts as Container) and the Instruction Class and the BooleanExpression Class of the right meta model as contained classes

See AssignmentDC meta model in Fig. 5.19.

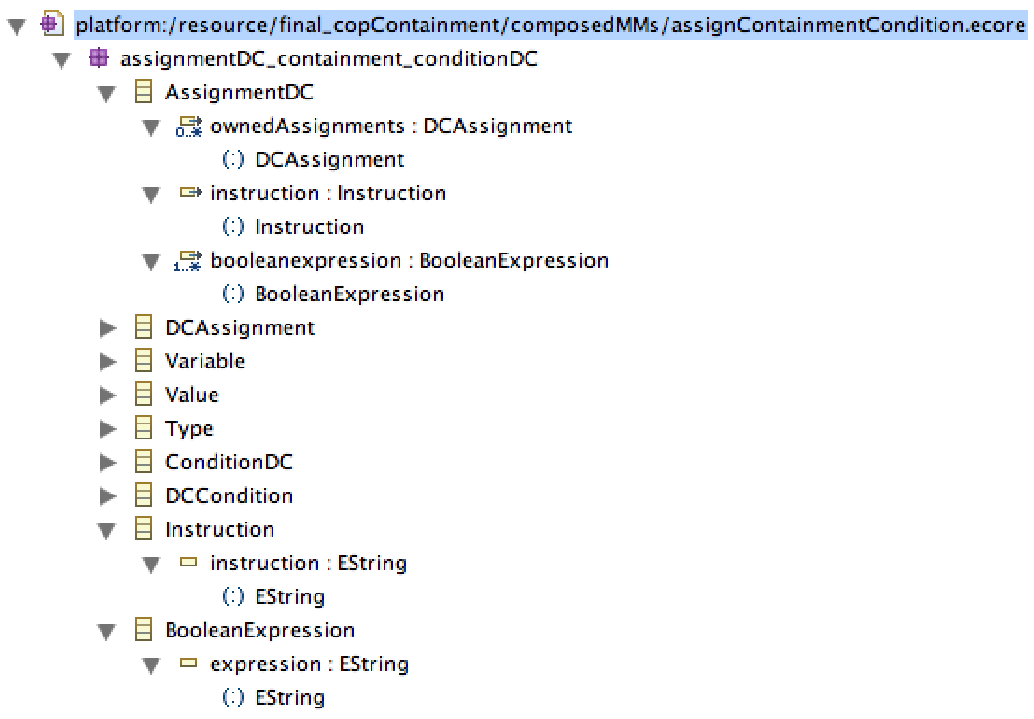
See ConditionDC meta model in Fig. 5.20.

See AssignmentDCContainementConditionDC meta model in Fig. 5.22.

So the question now, is how can we get from this specific ATL transformation example a Template for our Meta model Composition tool? We simply need to rewrite the EReference creation part of the 'CLASSRight2' ATL rule adapting it to the number and names of contained classes at one hand, to adapt also the 'CLASSRight1' rule and modify the endpoint rule setting the name of the requested Container class from the right meta model.

Please note that all the union problematic for this operator (same class names in left and right meta model for example) are not treated by us, it depends on ATL launcher and on the way they manage this type of problems.

For transformation of instance of meta models, the information we need to know is only the classes names concerned by the containmnent relation newly created. So the tool will generate a file with this useful information. For more details on instances transformation, please refer to the corresponding



```
▼ platform:/resource/final_copContainment/composedMMs/assignContainmentCondition.ecore
  ▼ assignmentDC_containment_conditionDC
    ▼ AssignmentDC
      ▼ ownedAssignments : DCAssignment
          DCAssignment
      ▼ instruction : Instruction
          Instruction
      ▼ booleanexpression : BooleanExpression
          BooleanExpression
    ▶ DCAssignment
    ▶ Variable
    ▶ Value
    ▶ Type
    ▶ ConditionDC
    ▶ DCCondition
    ▼ Instruction
      ▼ instruction : EString
          EString
    ▼ BooleanExpression
      ▼ expression : EString
          EString
```

Figure 5.22. AssignmentDCContainementConditionDC Ecore meta model, result of the ATL transformation

chapter.

ATL Transformation for Association Operator

The Association operator produces as result meta model a meta model with the content of left and right meta models and will add an association relation between a class of left meta model and a class of right meta model.

The ATL transformation for that is also easy to understand. In the 'init' rule we just create the resulting EPackage, set its name and copy in it the eClassifiers of left and right meta model.

Then, we simply create in the resulting meta model all the EClasses, EAttributes and EReferences of both meta models with ATL rules.

The ATL solution found for creating the association relation is the following. Left EClasses are normally copied to resulting meta model. But for classes from the right meta model, we defined two ATL rules called 'CLASS-Right1' and 'CLASSRight2'.

The first rule will take the class from right meta model that is the supplier class. For this class, we create the resulting EClass plus an EReference for the association relation. Why are we creating the eReference at this point? This is because when creating the containment eReference, we give it the name of the eClass, we set the lower and upper bounds, and the more important we set the eType of the association which is the freshly created EClass. We then also add the association eReference to a list because we will then need to add those new eReference to the supplied class from the left meta model. This is done in the delayed actions in the endpoint ATL rule.

The second rule ('CLASSRight2' ATL rule) will match all the classes from right meta model that are not the supplier element and simply create the resulting EClass in the composed meta model.

You can take a look to the Association Operator Template in Appendix A.

For a generic template, we only need to fix the name of the supplier and of the supplied classes.

Please note that all the union problematic for this operator (same class names in left and right meta model for example) are not treated by us, it depends on ATL launcher and on the way they manage this type of problems.

For transformation of instance of meta models, the information we need to know is only the classes names concerned by the containmnent relation newly created. So the tool will generate a file with this useful information. For more details on instances transformation, please refer to the corresponding chapter.

ATL Transformation for Union Operator

The Union operator produces as result meta model a meta model with the content of left and right meta models and will rename all the elements of right meta model that are already in left meta model.

The ATL transformation for that simply copy elements from left and right meta models and renames classes that need to be renamed. As endpoint rules we manage to rename and correct all the eAttributes, eReferences and eTypes that shoul be renamed.

You can take a look to the Union Operator Template in Appendix A.

For a generic template, nothing is needed.

For transformation of instance of meta models, the information we need to know is only the classes names concerned by the containmnent relation newly created. So the tool will generate a file with this useful information. For more details on instances transformation, please refer to the corresponding chapter.

ATL Transformation for Merge Operator

The Merge operator produces as result meta model a meta model with the content of left and right meta models and will merge all the elements of right meta model that are already in left meta model.

The ATL transformation for that simply copy elements from left and right meta models and merges classes that are already in left meta model. All the merged classes are added to a list. As endpoint rules we browse this list of merged classes in order to add to the resulting class the eAttributes, eReferences or eTypes that should be added from the right meta model class to the left meta model class.

You can take a look to the Merge Operator Template in Appendix A.

For a generic template, nothing is needed.

For transformation of instance of meta models, the information we need to know is only the classes names concerned by the containmnent relation newly created. So the tool will generate a file with this useful information. For more details on instances transformation, please refer to the corresponding chapter.

5.4.4 Implementation Details

The tool for meta model composition has been implemented with Java language. It's class diagram can be seen on Fig. 5.23

The MMComposition class has a switch case type of selection and will create an instance of the corresponding composition operator. The CompOperator abstract class is associated to an ATLCompiler and to an ATLLauncher in order to be able to compile the specific ATL file and to launch the ASM resulting transformation file for producing the expected output.

There are problems with the execution of the generated ASM compiled files. But if we take those ATL transformation and launch them into the Eclipse ATL Plugin, they are working.

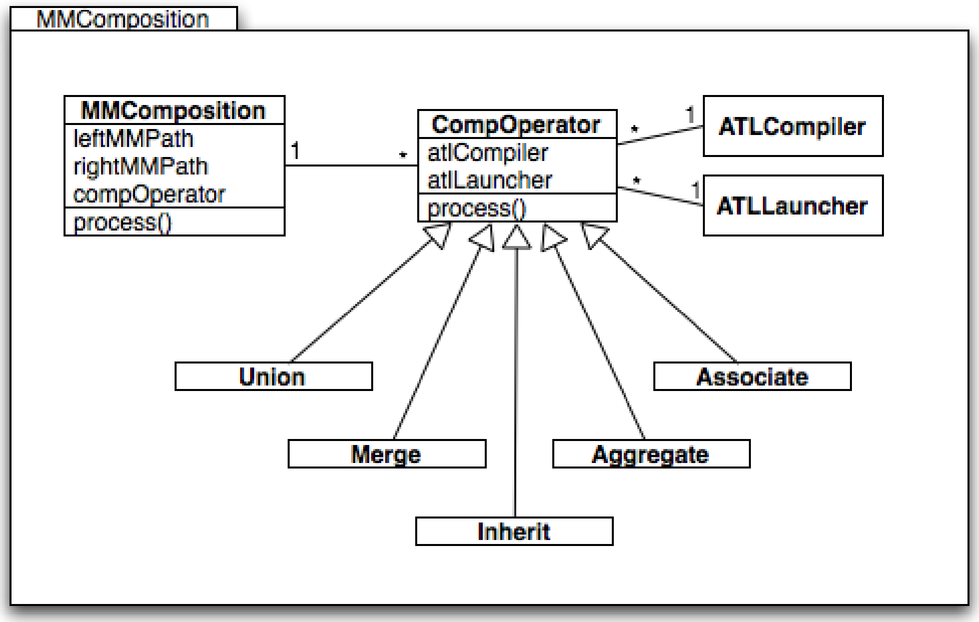


Figure 5.23. UML Class Diagram for Meta Model Composition Tool

Chapter 6

Composition of instances of Meta Model

In previous sections, we saw how we have implemented a tool for automatical generation of ATL transformation of meta models. In this chapter, we are going to present the conceptual approach and how we have implemented a module in our Meta Model Composition Tool that composes the instances of the meta models.

6.1 Global view

For computing the transformation of the instances of the input meta models in order to get as output an instance of the composed meta model, we need the following information as input:

- the left meta model path;
- the right meta model path;
- the operator;
- the left meta model instance path;
- the right meta model instance path;
- the composed meta model path;

So we will have to give as input to the tool the composition definition and the path to the resulting composed meta model.

With all that information, we will be able to process the following steps:

- 1 - Transform the left meta model instance to make them conform to the composed meta model;
- 2 - Transform the right meta model instance to make them conform to the composed meta model;
- 3 - Do the union of the two resulting composed meta model instances ;
- 4 - Apply the corresponding operator changes to the composed meta model instances;

For doing the two first steps, we then need to create ATL transformation specific to the corresponding meta models to which the models are conforming. This means that we need to find a solution for generating automatically a set of rules that will be used then to create an ATL model composition transformation.

We also need to get some information on which class names or attributes have been modified during the meta model composition (renaming of classes for examples as explained in Chapter V).

6.2 Meta Model for Specific Transformation Rules

In order to produce a file with the rules specific to the old and new meta models, we defined a meta model that will allow to store ATL rules. This meta model can be seen in Fig. 6.1.

The root is composed of Transformation Rules. In one hand we have all the rules for the left meta model, and in the other hand we have all the rules for the right meta model.

The ATL rules are stored as simple Strings. We are going to see how we generate those rules in the next section.

This allow to generate instances of this meta model during the meta model composition. The instance will be read during the model composition phase.

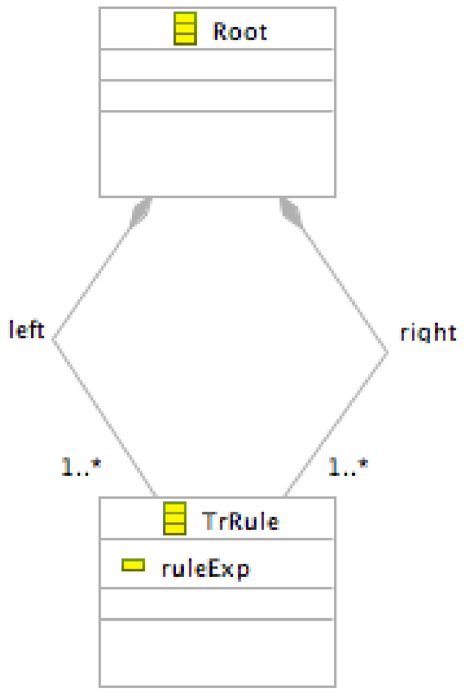


Figure 6.1. Transformation Rules Meta Model

6.3 Transformation Rules Generation

The generation of the transformation rules takes place during the meta model composition process. It takes an ATL template that will take as input the left and the right meta models and will produce as output an instance of the Transformation Rules meta model. This instance will contain the rules for left and right meta models. This file will be useful for the model composition as previously explained.

As it can be seen in the corresponding code listings in Appendix A, we need to specify the

- the package name from the left meta model (replace @@leftPackageName@@ string in template);

- the package name from the right meta model (replace @@rightPackageName@@ string in template);

- the class name from the root of the left meta model (replace @@ left-RootCName@@ string in template);
- the class name from the root of the right meta model (replace @@ rightRootCName@@ string in template);

Then, as it can be seen in Appendix, depending on the operator we will generate the rules in different ways. For example, for Merge and Union operators, we need to take in account the fact that some classes has been renamed or merged.

Another important thing to be said, is that the root element of the meta models should always be the first element.

6.4 Transformation of Instances to conform them to the composed Meta Model

In this step, we have 2 operations:

- 1 - Transform the left meta model instance to make them conform to the composed meta model;
- 2 - Transform the right meta model instance to make them conform to the composed meta model;

For the first point, it is quiet easy to do this transformation with ATL as it is a simple copy of elements as in the implementation of our composition operators we never change the naming of the elements of the left meta model.

But, for the second point, it is a bit more complicated as the name of the elements of the right meta model could have been renamed. For being able to deal with that problem, we need to know what classes and/or attributes have been changed during the composition transformation. The produced file with this information is the instance of the Transformation Rules seen in previous section.

6.5 Union of the Instances

Once we have adapted the two instances in order them to conform to the new composed meta model, we need to do an union of them. This union is also done when generating the Transformation Rules as described in previous chapter.

6.6 Apply the Operator changes in the Composed Meta Model Instance

Now that we have our composed meta model instance, the last step is to modify it depending on the used composition operator. This means that we need to create if needed the relations between some elements of the instance depending on the chosen meta model composition operator.

To do that, we are now faced to some questions. For example, if we have created an association in the composed meta model between to classes, what should we do with the elements instances of these classes in the composed model? Should we keep them separated? Should we create a link between them? What happen if we have cardinalities issues?

This problem has been solved taking the decision to give the user three distinct possibilities.

- 1 - Keep it separated;
- 2 - Do a unique and random relation;
- 3 - Do random relations depending on cardinalities;

If any rule is not respected, the model will not validate in Eclipse.

For doing this, we have implemented for each operator a class in our Meta Model Composition Tool that will manage the model composition for its operator. Those classes, as done for meta model composition (presented in previous Chapter), have a process() method. This method reads a template for execution of model composition.

This template can be seen in Annex A. Basically, it has two variables that we search for and replace:

- @@modelTransformationRules@@;
- @@operatorRules@@;

For the first one, we load and read the Transformation Rules instance generated during the meta model composition and we just copy the rules in the new ATL file for model composition execution.

The second one, depends on the chosen operator and on the option (presented previously) taken. There is an if statement in each operator depending class. The ATL rules are presented in following subsections.

6.6.1 Association

For association, we need to add an association between each instance of the Suplier meta model class and of instance of the Client meta model class. Here are the different rules for the option filling all random and filling one random.

Listing 6.1. ATL rule for Association. Option filling all random.

```
...

for(dta in ComposedMM!@@clientClassName@@.allInstances()) {
  for (dtb in ComposedMM!@@suplierClassName@@.allInstances()) {
    dta.@@suplierClassNameInLowerCase@@<-dtb;
  }
}

...
```

Listing 6.2. ATL rule for Association. Option filling one random.

```
...

for(dta in ComposedMM!@@clientClassName@@.allInstances()) {
  dta.@@suplierClassNameInLowerCase@@<-ComposedMM!
      @@suplierClassName@@.allInstances()->selectAny(e | true);
}

...
```

6.6.2 Containment

For containment, we need to add the containment relations between the container and the contained instances.

Listing 6.3. Java code for ATL rule generation for Containment.

```
...

```

```
switch(fillingOption) {
    case 1:
            // atl rule for random filling
            operatorRules="\t\tfor(dta in ComposedMM!"+containerClassName+".allInstances()) {\n";
            Iterator containedClassesIterator = containedClasses.iterator();
            while(containedClassesIterator.hasNext()) {
                    ContainedClass currentContainedClass = (ContainedClass) containedClassesIterator.next();
                    operatorRules+="\t\t\tfor (dtb in ComposedMM!"+currentContainedClass.getClassName()+".allInstances()) {\n" +"\t\t\t\tdta."+currentContainedClass.getClassName().toLowerCase()+"<-dtb;\t\t\t\t\n" +"\t\t\t}\n" ;
            }
            operatorRules+="\t\t}";
            break;
    case 2:
            // atl rule for unique random filling
            operatorRules="\t\tfor(dta in ComposedMM!"+containerClassName+".allInstances()) {\n";
            Iterator containedClassesIterator2 = containedClasses.iterator();
            while(containedClassesIterator2.hasNext()) {
               ContainedClass currentContainedClass = (ContainedClass) containedClassesIterator2.next();
               operatorRules+="\t\t\tdta."+currentContainedClass.getClassName().toLowerCase()+" <- ComposedMM!"+currentContainedClass.getClassName()+".allInstances()->any(e | true);\n";
            }
            operatorRules+="\t\t}";
            break;
    default:
            // atl rule for not filling
            operatorRules="";
            break;
}

...
```

6.6.3 Inherit, Union and Merge

We do not need to apply any rule for those operators as the resulting models are disjunct.

Chapter 7

Case study

This chapter presents all the steps to be taken in order to realize a composition of meta models and of their instances through an example. All the steps are explained and illustrated with screenshots.

The example that will be taken into consideration in this case study is the composition of the Assignment and Condition Domain Concepts meta models (presented in Chapter IV). We are going to compose them with the Inherit composition operator in order to get a new meta model in which a possible instruction of the Condition statement can be an Assignment. This means that the Instruction class of the Condition Domain Concept meta model will be the specialized class. And the specialization class will be the DCAssignment class from the Assignment Domain Concept meta model.

7.1 Introduction

The meta models we are going to compose are:

- the Assignment Domain Concept meta model (In Fig. 4.1, Chapter IV);
- the Condition Domain Concept meta model (In Fig. 4.3, Chapter IV);

The operator chosen for the composition is the Inherit operator. It will take:

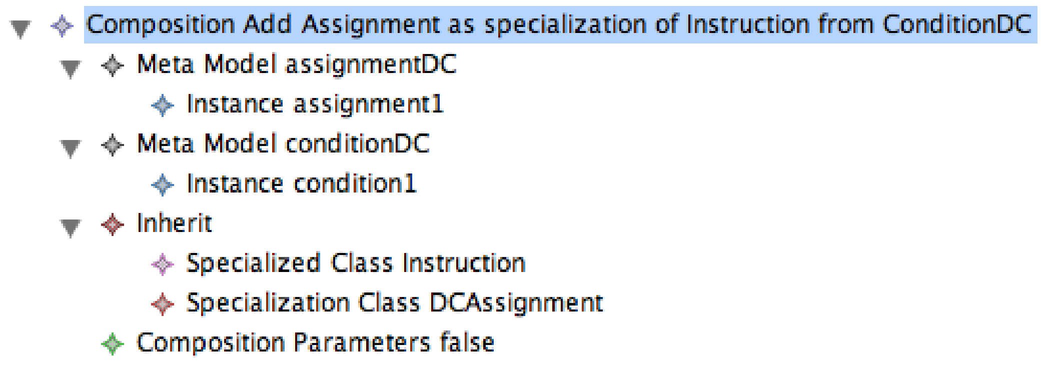
- as specialized class: Instruction class from Condition Domain Concept meta model;

- as specialization class: DCAssignment class from Assignment Domain Concept meta model;

We have also defined one instance per meta model. The definitions of those meta models have been done using the EMF Eclipse framework. The instances of the meta models have been done using the editors generated automatically by the EMF Eclipse framework. More information on those steps can be found on EMF Eclipse website.

7.2 Defining a composition

Now that we have defined the inputs and the operator, we launch the editor generated automatically by the EMF Eclipse framework for our meta model for meta model composition (described in Chapter V, see Fig. 5.1).



```
▼ Composition Add Assignment as specialization of Instruction from ConditionDC
    ▼ Meta Model assignmentDC
          Instance assignment1
    ▼ Meta Model conditionDC
          Instance condition1
    ▼ Inherit
          Specialized Class Instruction
          Specialization Class DCAssignment
      Composition Parameters false
```

Figure 7.1. Definition of the composition

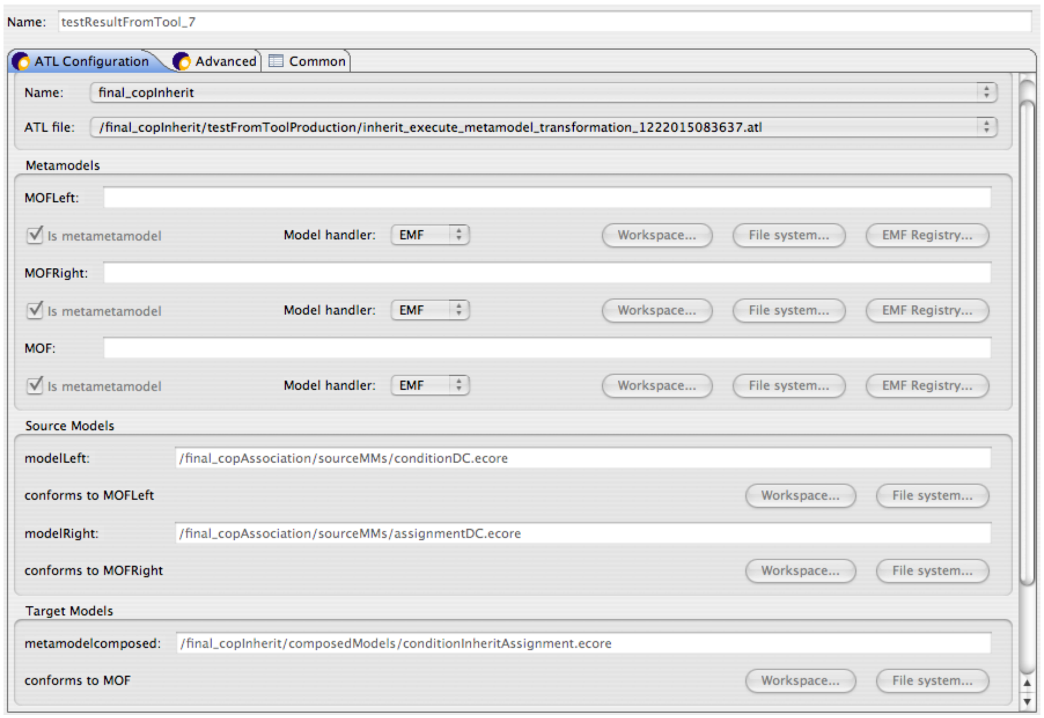
The Fig. 7.1 show the meta model instance we have created for the example we have taken into consideration.

7.3 Execution of the Meta Model Composition

As described in Chapter V, the Meta Model Composition Tool I developed take as input the composition definition file. You just need to execute the tool, and you will get as result an ATL transformation file containing the rules for getting the expected resulting meta model.

You will have to take this produced file and run it in the ATL plugin in the Eclipse framework as the interface for programmatical launch still doesn't work.

Then, in ATL you have to define the run configuration and you will get the expected composed meta model. You can find the screenshot of the ATL run configuration for executing the meta model composition in Fig. 7.2.



Name: testResultFromTool_7

Name: final_copInherit

ATL file: /final_copInherit/testFromToolProduction/inherit_execute_metamodel_transformation_1222015083637.atl

Metamodels

MOFLeft:

✓ Is metametamodel Model handler: EMF

MOFRight:

✓ Is metametamodel Model handler: EMF

MOF:

✓ Is metametamodel Model handler: EMF

Source Models

modelLeft: /final_copAssociation/sourceMMs/conditionDC.ecore

conforms to MOFLeft

modelRight: /final_copAssociation/sourceMMs/assignmentDC.ecore

conforms to MOFRight

Target Models

metamodelcomposed: /final_copInherit/composedModels/conditionInheritAssignment.ecore

conforms to MOF

Figure 7.2. Screenshot of the ATL Run Configuration in Eclipse for running the ATL meta model composition produced file.

The resutling meta model is in Fig. 7.3.

As it can be seen, we have in the left part the Condition Domain Concept Meta Model and in the Right part the Assignment Domain Concept Meta Model with a new inheritance relation that signifies that an Assignment can be an instruction of a Condition. That was the expected result. The meta model also validates in the Eclipse Framework, which is a good warranty for it correctness.

We can then generate the editor automatically with the EMF Eclipse Plugin in order to create new instances of the meta model. You can find in

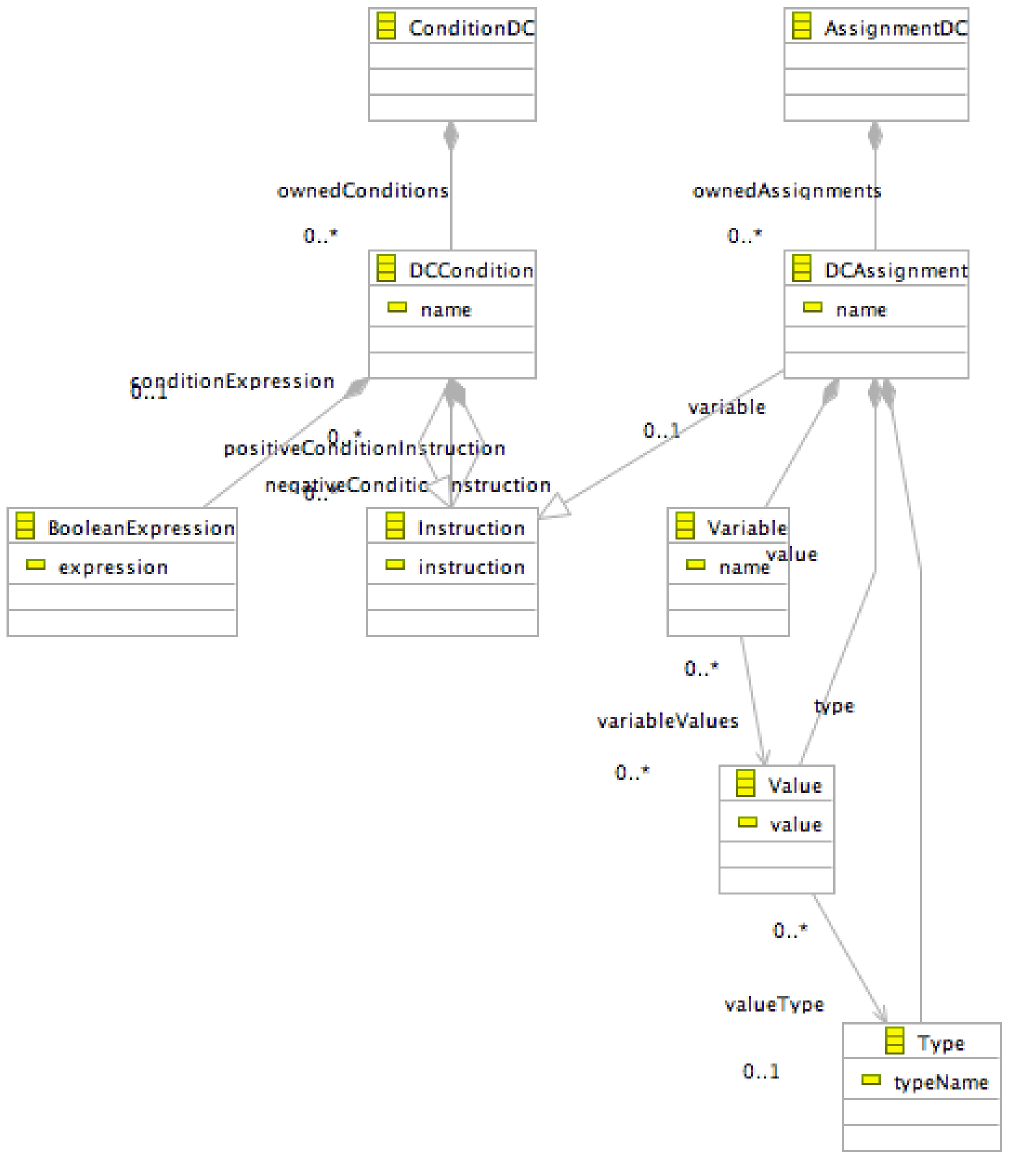


Figure 7.3. Meta Model resulting from the composition. Operator: Inherit. Source meta models: Condition DC and Assignment DC.

Fig. 7.4 an example of an instance of our new composed meta model.

This model validates in the Eclipse Framework.

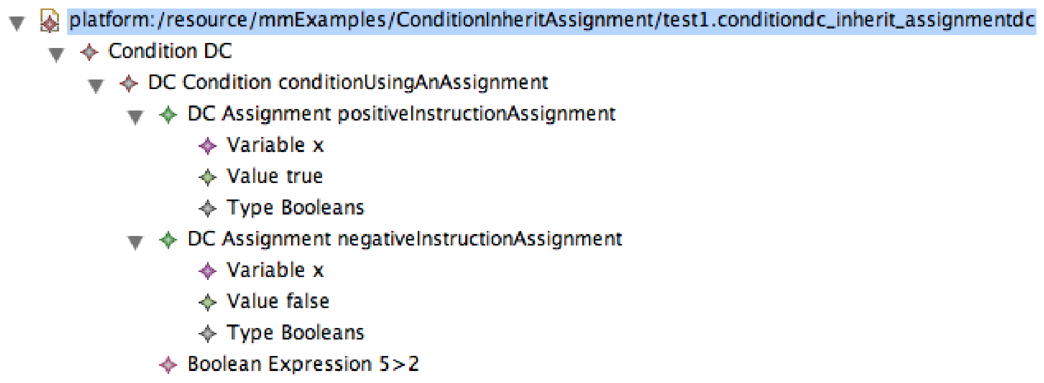

platform:/resource/mmExamples/ConditionInheritAssignment/test1.conditiondc_inherit_assignmentdc
Condition DC
DC Condition conditionUsingAnAssignment
DC Assignment positiveInstructionAssignment
Variable x
Value true
Type Booleans
DC Assignment negativeInstructionAssignment
Variable x
Value false
Type Booleans
Boolean Expression 5>2

Figure 7.4. Tree view of a new instance of the composed meta model.

7.4 Generation of Model Composition Rules

While executing the meta model composition, another ATL file is generated. This ATL file contents all the rules needed for being able to generate a composition of the instances of the meta models.

You can find in Fig. 7.5 a screenshot of the ATL run configuration for this example.

7.5 Execution of Composition of Instances of Meta Models

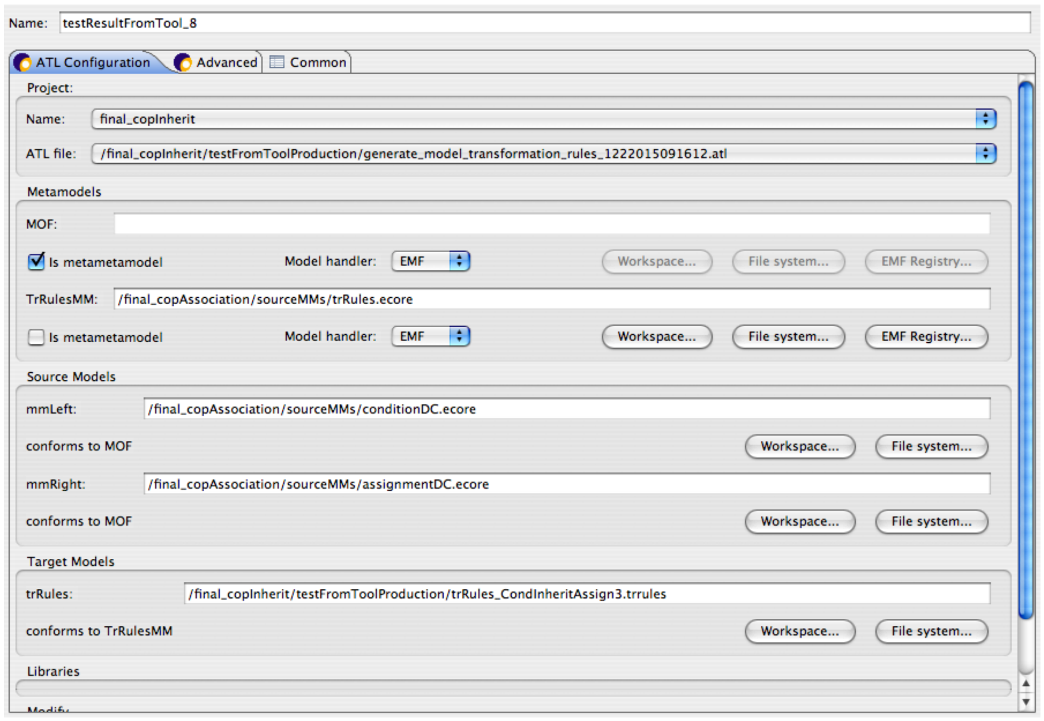
In order to compose the instances of the meta models of the meta model composition, we just need to execute again the Meta Model Composition Tool (described in Chapter V) giving as inputs the definition of the composition and the composed meta model path.

The tool will then generate an ATL file with the model transformation rules. This file can be seen in code listing following.

Listing 7.1. ATL automatically generated file for model composition execution.

```
module execute_model_transformation; -- Module Template
create newModel : ComposedMM from modelLeft : MMLeft, modelRight
    : MMRight;

-- INITIALIZATION
-- simple copy of the elements of left and right models
```


Name: testResultFromTool_8

ATL Configuration | Advanced | Common

Project:
Name: final_copInherit
ATL file: /final_copInherit/testFromToolProduction/generate_model_transformation_rules_1222015091612.atl

Metamodels
MOF:
[✓] Is metamodel Model handler: EMF
TrRulesMM: /final_copAssociation/sourceMMs/trRules.ecore
[ ] Is metamodel Model handler: EMF

Source Models
mmLeft: /final_copAssociation/sourceMMs/conditionDC.ecore
conforms to MOF
mmRight: /final_copAssociation/sourceMMs/assignmentDC.ecore
conforms to MOF

Target Models
trRules: /final_copInherit/testFromToolProduction/trRules_CondInheritAssign3.trrules
conforms to TrRulesMM

Libraries

Figure 7.5. Screenshot of the ATL Run Configuration in Eclipse for running the ATL model composition transformation rules produced file.

```
-- to the new model conforming to composed meta model

 rule L_ConditionDC { from pack : MMLeft!"ConditionDC" to
     newlPack: ComposedMM!"ConditionDC"( ownedConditions <- pack.
     ownedConditions->collect(e | thisModule.L_DCCondition (e) )
     ) }
 lazy rule L_DCCondition { from pack : MMLeft!"DCCondition" to
     newlPack: ComposedMM!"DCCondition"( name <- pack.name,
     positiveConditionInstruction <- pack.
     positiveConditionInstruction, negativeConditionInstruction
     <- pack.negativeConditionInstruction, conditionExpression
     <- pack.conditionExpression) }
 rule L_Instruction { from pack : MMLeft!"Instruction" to
     newlPack: ComposedMM!"Instruction"( instruction <- pack.
     instruction) }
 rule L_BooleanExpression { from pack : MMLeft!"
     BooleanExpression" to newlPack: ComposedMM!"
     BooleanExpression"( expression <- pack.expression) }
 rule R_AssignmentDC { from pack : MMRight!"AssignmentDC" to
    newrPack: ComposedMM!"AssignmentDC"( ownedAssignments <-
    pack.ownedAssignments->collect(e | thisModule.R_DCAssignment
     (e) )) }
 lazy rule R_DCAssignment { from pack : MMRight!"DCAssignment"
    to newrPack: ComposedMM!"DCAssignment"( name <- pack.name,
    variable <- pack.variable, value <- pack.value, type <-
    pack.type) }
 rule R_Variable { from pack : MMRight!"Variable" to newrPack:
    ComposedMM!"Variable"( name <- pack.name, variableValues <-
     pack.variableValues) }
 rule R_Value { from pack : MMRight!"Value" to newrPack:
    ComposedMM!"Value"( value <- pack.value, valueType <- pack.
    valueType) }
 rule R_Type { from pack : MMRight!"Type" to newrPack:
    ComposedMM!"Type"( typeName <- pack.typeName) }


-- EXECUTION OF DELAYED ACTIONS
-- rules depending on the operator

-- rules for association
endpoint rule EndRule() {
  do {
    -- rule for operator Inherit
    -- No Rule Needed
  }
}
```

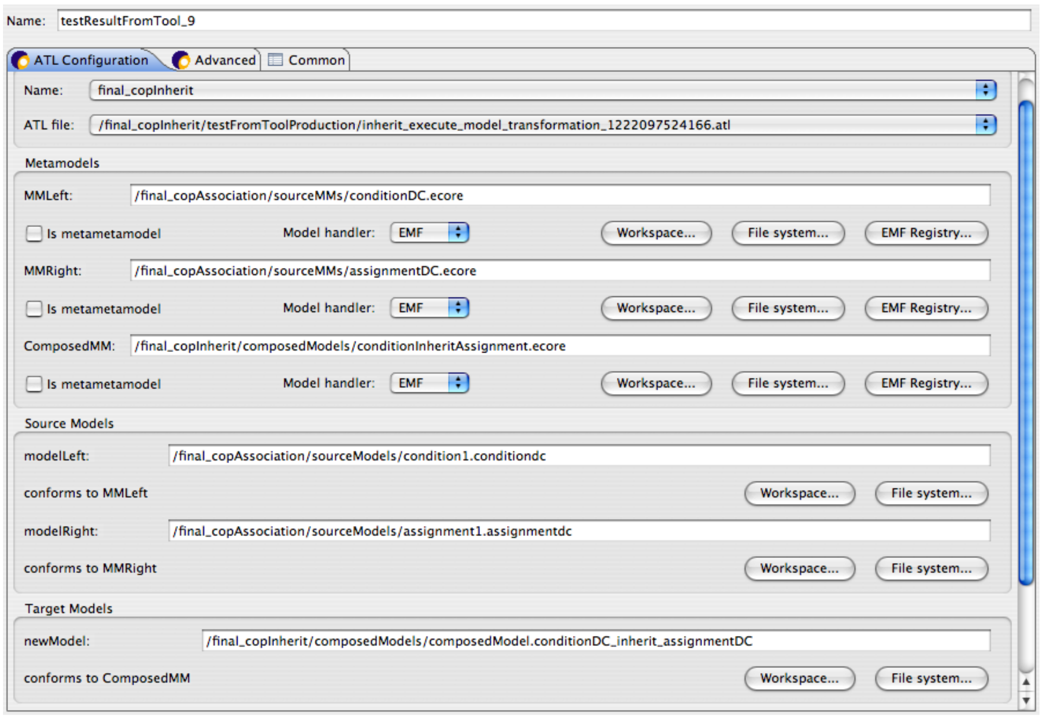
You can find in Fig. 7.6 a screenshot of the ATL run configuration for this example.

The resulting composed model can be seen in Fig. 7.7.

This model validates in the Eclipse Framework.

7.6 Meta model checking and verification

As described on Chapter IV, we can transform a meta model into a COOPN specification. An idea for further work would be to analyze how we can compose the defined transformation of Domain Concepts into COOPN, in order to get the COOPN specification of the resulting meta model. This will allow to do some meta model checking and verifications.


Name: testResultFromTool_9

Name: final_copInherit

ATL file: /final_copInherit/testFromToolProduction/inherit_execute_model_transformation_1222097524166.atl

Metamodels

MMLeft: /final_copAssociation/sourceMMs/conditionDC.ecore

☐ Is metametamodel Model handler: EMF

MMRight: /final_copAssociation/sourceMMs/assignmentDC.ecore

☐ Is metametamodel Model handler: EMF

ComposedMM: /final_copInherit/composedModels/conditionInheritAssignment.ecore

☐ Is metametamodel Model handler: EMF

Source Models

modelLeft: /final_copAssociation/sourceModels/condition1.conditiondc

conforms to MMLeft

modelRight: /final_copAssociation/sourceModels/assignment1.assignmentdc

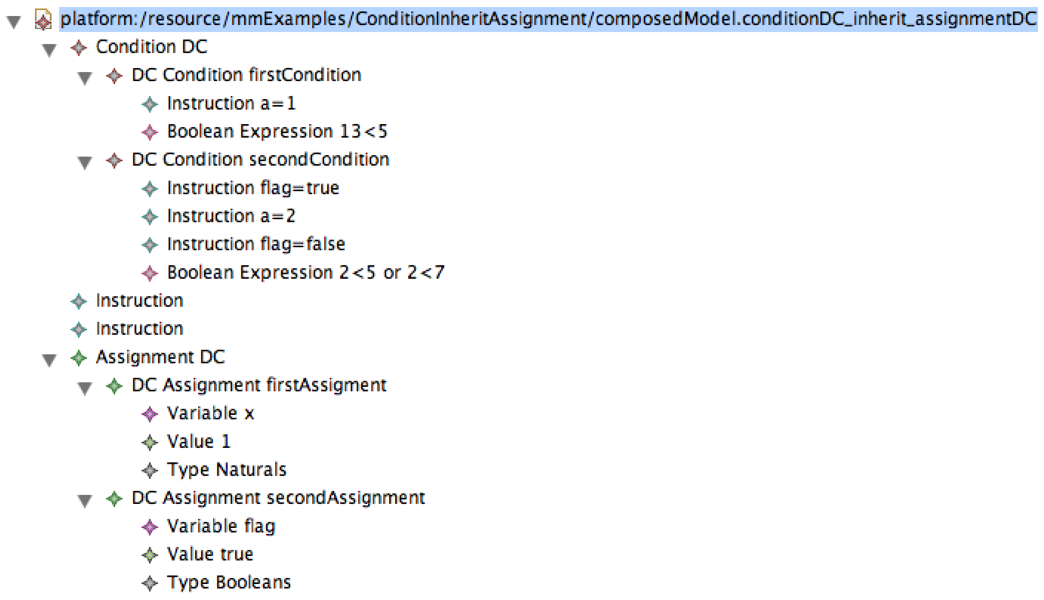
conforms to MMRight

Target Models

newModel: /final_copInherit/composedModels/composedModel.conditionDC_inherit_assignmentDC

conforms to ComposedMM

Figure 7.6. Screenshot of the ATL Run Configuration in Eclipse for running the ATL model composition produced file.



```
▼ platform:/resource/mmExamples/ConditionInheritAssignment/composedModel.conditionDC_inherit_assignmentDC
  ▼ Condition DC
    ▼ DC Condition firstCondition
        Instruction a=1
        Boolean Expression 13<5
    ▼ DC Condition secondCondition
        Instruction flag=true
        Instruction a=2
        Instruction flag=false
        Boolean Expression 2<5 or 2<7
    Instruction
    Instruction
  ▼ Assignment DC
    ▼ DC Assignment firstAssigment
        Variable x
        Value 1
        Type Naturals
    ▼ DC Assignment secondAssignment
        Variable flag
        Value true
        Type Booleans
```

Figure 7.7. Tree view of resulting composed model.

Chapter 8

Conclusion

In this Master work, we have

- studied the meta model composition and its operator;
- implemented the meta model composition operators with ATL transformation language;
- studied and implemented the problematic of the composition of instances of meta models;
- seen how we can define domain concepts meta models which are bricks of programation languages;
- developped tools for generic meta models and instances composition;
- how we can transform those meta model in their equivalent in COOPN language for testing and verification aims;
- how we can trasnform instances of the COOPN language meta model into COOPN Specifications;

This Master work have been the fruit of a long work. I appreciate that through this Master I have increased my knowledges about meta models, about COOPN language, and specially about meta model and model transformation with the ATL language.

We saw with the case study, that in theory very intersting things can be done with meta model composition. But one thing I can see is that for the moment the ATL programmatically launch is still a problem as I did not managed to have it work. This was not very good for developing the composition tools, as I should always take generated ATL files to launch them into the Eclipse ATL Plugin.

Another thing that appears, is the fact that for instances of meta model composition, a lot of interesting questions appears depending on the operators. More work should be done on this subject.

Finally, all the verification and test side of meta models with COOPN language and the composition of their transformation is still a main research field that should be studied.

The conclusion of this Master work is that work has been done on the meta model composition, but this opens now a lot of new subjects and questions that should be explored in further works.

Appendix A

Implementation

This chapter makes a presentation of the main implementation aspects.

A.1 XMI to COOPN Transformation

Listing A.1. Recursive method for processing the equations for COOPN condition, axioms and theorems.

```
private static void processEquation(Equation theEquation){
    // Getting left and right terms of the equation
    Term leftTerm = theEquation.getOwnedOperatorEquation().
        getLeftTerm();
    Term rightTerm =  theEquation.getOwnedOperatorEquation().
        getRightTerm();
    // For each term of the equation
    for (Object aTerm : theEquation.getOwnedEquationTerms()){
        Term term = (Term)aTerm;
        // if the term is not an equation, that means that we
            are on the recursion terminal case
        if(!(aTerm instanceof Equation)){
            // writing the current term
            equationExpression += " " +  term.getExpression() +
                " ";
            // and if there is a right term we write the
                operator
            if(aTerm == leftTerm){
                equationExpression+= theEquation.
                    getOwnedOperatorEquation().getOperator() + "
                     " ;
            }
        }
        // if the term is an equation, we will have to do a
            recursion step
        if(aTerm instanceof Equation){
            Equation equation = (Equation)aTerm;
            if (!(equation.getOwnedOperatorEquation().
                getLeftTerm() instanceof Equation)){
                // if left term not an equation, we can write
                    it
                equationExpression += "(" + equation.
                    getOwnedOperatorEquation().getLeftTerm().
                    getExpression() ;
            }
            else{
                // if left term an equation, we can proceed to
                    recursion
                processEquation(equation);
            }
            // adding the operator
            equationExpression += " " + equation.
                getOwnedOperatorEquation().getOperator() + " ";
            if (!(equation.getOwnedOperatorEquation().
                getRightTerm() instanceof Equation)){
                // if right term not an equation, we can write
                    it
                equationExpression += equation.
                    getOwnedOperatorEquation().getRightTerm().
                    getExpression() + ")";
                // adding operator
                if(aTerm == leftTerm){
                    equationExpression+= " " + theEquation.
                        getOwnedOperatorEquation().getOperator()
                         + " " ;
                }
            }
            else{
                // if right term is an equation, we can proceed
                     to recursion
                processEquation(equation);
            }
        }
}
```

A.2 ATL Operator Templates

You can find here the code listings for the ATL Templates for each meta model composition operator.

A.2.1 Template for Union Operator

Listing A.2. ATL Template Transformation for Union operator

```
-- @atlcompiler atl2006
module union_execute_metamodel_transformation; -- Module
    Template
create metamodelcomposed : MOF from modelLeft : MOFLeft,
    modelRight: MOFRight;

-- HELPING METHODS

-- method for getting a collection of all the class names from
    the left package
helper context MOFLeft!EPackage def : getAllClassesNames () :
    OrderedSet(String) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(String)
      =
     OrderedSet{} | elements.append(c.name)
     )
  ;

-- definition of a collection of EReferences which will be used
-- for modifying eReferences types at the end
helper def: referencesFromResultMM : OrderedSet(MOF!EReference)
    =
  OrderedSet {};

-- definition of a collection of eReferences names from left
    metamodel
helper def: referencesNamesFromLMM : OrderedSet(String) =
  OrderedSet {};

-- definition of a collection with names of the classes from the
     right meta model
helper def: classesFromRMM : OrderedSet(String) =
  OrderedSet {};

-- definition of a collection of EAttributes which will be used
```

```
helper def: attributesFromResultMM : OrderedSet(MOF!EAttributes)
    =
  OrderedSet {};

-- definition of a collection of ESuperTypes which will be used
helper def: classesThatHaveSuperTypes : OrderedSet(MOFRigth!
    EClass) =
  OrderedSet {};

-- INITIALIZATION

rule init {
  from
    lPack : MOFLeft!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    compPack: MOF!"ecore::EPackage"(
      -- setting name of the new epackage
      name <- lPack.name + '_union_' + rPack.name,
      -- setting the eclassifies from left and right meta model
      eClassifiers <- lPack.eClassifiers,
      -- as the eClasses from the right meta model need to be
          renamed
      -- in case there is already a class with same name in the
          left package
      -- we need to use a lazy rule because we need to use the
          left epackage to check
      -- if a right mm class name is already used in the left mm
      eClassifiers <- rPack.eClassifiers->collect(e | thisModule
          .CLASSRight(e,lPack,compPack,rPack))
    )
}

-- MANAGING ECLASSES

rule CLASSLeft {
  from
    l : MOFLeft!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

lazy rule CLASSRight{
  from
    r : MOFRight!"ecore::EClass",
    l : MOFLeft!"ecore::EPackage",
    c : MOF!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    comp: MOF!"ecore::EClass"(
        -- we need lazy rules for attributes and references as
            we need to rename them
        eStructuralFeatures <- r.eAttributes,
        eStructuralFeatures <- r.eReferences->collect(e |
            thisModule.REFRight(e,l,c,rPack)),
        -- renaming the class name if the name is already used
            in left mm
        name <-    if l.getAllClassesNames().excludes(r.name)
                     then
                       r.name
                     else
                       r.name + '_RMM'
                   endif
    )
  do {
    -- feeling the collection of eClasses of the resulting mm
    thisModule.classesFromRMM2List(comp);
    if(r.eSuperTypes.size()>0) {
      thisModule.classesThatHaveSuperTypes2List(r);
    }
  }
}

-- MANAGING EATTRIBUTES

rule ATRLeft {
  from
    l : MOFLeft!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- eattributes from left mm just need to be put in the
          result mm
      name <- l.name,
      eType <- l.eType
    )
}

rule ATRRight {
  from
    r : MOFRight!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- right attributes need to be renamed and the type
          reference to be changed
      -- if pointing to a class that has been renamed
      name <-  r.name,
      eType <- r.eType
    )
  do {
    -- feeling the collection of ereferences
    thisModule.AttributesToList(comp);
  }
}

-- MANAGING EREFRENCES

rule REFLeft {
  from
    l : MOFLeft!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
          result mm
      name <-  l.name,
      upperBound <- l.upperBound,
      eType <- l.eType,
      containment <- l.containment
    )
  do {
    -- feeling the collection of ereferencesNames for renaming
        of eref names
    thisModule.ReferencesNamesToList(comp);
  }
}

lazy rule REFRight {
  from
    r : MOFRight!"ecore::EReference",
    l : MOFLeft!"ecore::EPackage",
    c  : MOF!"ecore::EPackage",
    rPack  :  MOFRight!"ecore::EPackage"
  to
    comp: MOF!"ecore::EReference"(
      -- right mm eReferences need to be changed if pointing to
          a class that has been renamed
      -- but now we just copy them, as the modifications can
          only be done at the end
      name <-   r.name,
      lowerBound <- r.lowerBound,
      upperBound <- r.upperBound,
      eType <- r.eReferenceType,
      containment <- r.containment
    )
  do {
    -- feeling the collection of ereferences
    thisModule.ReferencesToList(comp);
  }
}

-- FEELING COLLECTION RULES

rule ReferencesToList (e : MOF!EReference) {
  do {
    thisModule.referencesFromResultMM <- thisModule.
        referencesFromResultMM->append(e);
  }
}

rule ReferencesNamesToList (e : MOF!EReference) {
  do {
    thisModule.referencesNamesFromLMM <- thisModule.
        referencesNamesFromLMM->append(e.name);
  }
}

rule classesFromRMM2List (e : MOFRigth!EClass) {
  do {
    thisModule.classesFromRMM <- thisModule.classesFromRMM->
        append(e.name);
  }
}

rule AttributesToList (e : MOF!EAttributes) {
  do {
    thisModule.attributesFromResultMM <- thisModule.
        attributesFromResultMM->append(e);
  }
}

rule classesThatHaveSuperTypes2List (e : MOFRigth!EClass) {
  do {
    thisModule.classesThatHaveSuperTypes <- thisModule.
        classesThatHaveSuperTypes->append(e);
  }
}

-- EXECUTION OF DELAYED ACTIONS

endpoint rule EndRule() {
  do {
    -- for the inheritances relation adaptation
    for(dta in thisModule.classesThatHaveSuperTypes) {
      for(dtb in dta.eSuperTypes) {
        MOF!EClass.allInstances()->any(e | e.name = dta.name or
            e.name = (dta.name + '_RMM')).eSuperTypes <-
         if (thisModule.classesFromRMM.includes(dtb.name))
              then
                (
                (MOF!EClass.allInstances()->any(e | e.name = dtb.
                    name))
                )
              else
                (
                (MOF!EClass.allInstances()->any(e | e.name = (dtb.
                    name + '_RMM')))
                )
            endif;
      }
    }
    -- renaming and modifying eAttributes
    for(dta in thisModule.attributesFromResultMM) {
      dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eType.name))
              then
                (
                (MOF!EClass.allInstances()->any(e | e.name = dta.
                    eType.name))
                )
              else
              (
              (MOF!EClass.allInstances()->any(e | e.name = (dta.
                 eType.name + '_RMM')))
              )
           endif;
     }
     -- renaming and modifying eReferences
     for(dta in thisModule.referencesFromResultMM) {
       -- we can now rename the eReference name
       dta.name <- if(thisModule.referencesNamesFromLMM.includes(
          dta.name))
              then
                (dta.name+'_RMM')
              else
                (dta.name)
            endif;
       -- and modify the binding to the good EClass
       dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eReferenceType.name))
              then
                (
                 (MOF!EClass.allInstances()->any(e | e.name =
                    dta.eReferenceType.name))
                )
              else
                (
                 (MOF!EClass.allInstances()->any(e | e.name = (
                    dta.eReferenceType.name + '_RMM')))
                )
            endif;
     }
  }
}
```

A.2.2 Template for Merge Operator

Listing A.3. ATL Template Transformation for Merge operator

```
-- @atlcompiler atl2006
module merge_execute_metamodel_transformation; -- Module
    Template
create metamodelcomposed : MOF from modelLeft : MOFLeft,
    modelRight: MOFRight;

```

```
-- HELPING METHODS

-- method for getting a collection of all the class names from
    the left package
helper context MOFLeft!EPackage def : getAllClassesNames () :
    OrderedSet(String) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(String)
      =
   OrderedSet{} | elements.append(c.name)
   )
  ;

-- definition of a collection of EReferences which will be used
-- for modifying eReferences types at the end
helper def: referencesFromResultMM : OrderedSet(MOF!EReference)
    =
  OrderedSet {};

-- helpers in order to be able to access directly the left and
    right EPackages
helper def: thePackageFromRightMM : MOFRight!EPackage = MOFRight
    !EPackage;
helper def: thePackageFromLeftMM : MOFLeft!EPackage = MOFLeft!
    EPackage;

-- definition of a collection of eReferences names from left
    metamodel
helper def: referencesNamesFromLMM : OrderedSet(String) =
  OrderedSet {};

-- definition of a collection with names of the classes from the
     right meta model
helper def: classesFromRMM : OrderedSet(String) =
  OrderedSet {};

-- definition of a collection of EAttributes which will be used
helper def: attributesFromResultMM : OrderedSet(MOF!EAttributes)
     =
  OrderedSet {};

-- definition of a collection of ESuperTypes which will be used
helper def: classesThatHaveSuperTypes : OrderedSet(MOFRigth!
    EClass) =
  OrderedSet {};

```

```
-- INITIALIZATION

rule init {
  from
    lPack : MOFLeft!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    compPack: MOF!"ecore::EPackage"(
      -- setting name of the new epackage
      name <- lPack.name + '_merge_' + rPack.name,
      -- setting the eclassifies from left and right meta model
      eClassifiers <- lPack.eClassifiers ,
      -- as the eClasses from the right meta model need to be
          renamed
      -- in case there is already a class with same name in the
          left package
      -- we need to use a lazy rule because we need to use the
          left epackage to check
      -- if a right mm class name is already used in the left mm
      eClassifiers <- (rPack.eClassifiers->select(e | lPack.
          getAllClassesNames().excludes(e.name)))->collect(e |
          thisModule.CLASSRight(e,lPack,compPack,rPack)
    )
  do{
    thisModule.thePackageFromRightMM<-rPack;
    thisModule.thePackageFromLeftMM<-lPack;
  }
}

-- MANAGING ECLASSES

rule CLASSLeft {
  from
    l : MOFLeft!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      eSuperTypes <- l.eSuperTypes,
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
      eStructuralFeatures <- l.eStructuralFeatures
    )
  do {
    -- feeling the collection of eClasses of the resulting mm
    thisModule.classesFromRMM2List(comp);
  }
}

lazy rule CLASSRight{
  from
    r : MOFRight!"ecore::EClass",
    l : MOFLeft!"ecore::EPackage",
    c : MOF!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    comp: MOF!"ecore::EClass"(
        -- we need lazy rules for attributes and references as
           we need to rename them
        eStructuralFeatures <- r.eAttributes->collect(e |
           thisModule.ATRRight(e)),
        eStructuralFeatures <- r.eReferences->collect(e |
           thisModule.REFRight(e,l,c,rPack)),
        -- renaming the class name if the name is already used
           in left mm
        name <-    if l.getAllClassesNames().excludes(r.name)
                     then
                       r.name
                     else
                       r.name + '_RMM'
                   endif
    )
  do {
    -- feeling the collection of eClasses of the resulting mm
    thisModule.classesFromRMM2List(comp);
    if(r.eSuperTypes.size()>0) {
      thisModule.classesThatHaveSuperTypes2List(r);
    }
  }
}

-- MANAGING EATTRIBUTES

rule ATRLeft {
  from
    l : MOFLeft!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- eattributes from left mm just need to be put in the
         result mm
      name <-  l.name,
      eType <- l.eType
    )
}

lazy rule ATRRight {
  from
    r : MOFRight!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- right attributes need to be renamed and the type
          reference to be changed
      -- if pointing to a class that has been renamed
      name <-  r.name,
      eType <- r.eType
    )
  do {
    -- feeling the collection of ereferences
    thisModule.AttributesToList(comp);
  }
}

-- MANAGING EREFRENCES

rule REFLeft {
  from
    l : MOFLeft!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
          result mm
      name <-  l.name,
      upperBound <- l.upperBound,
      eType <- l.eType,
      containment <- l.containment
    )
  do {
    -- feeling the collection of ereferencesNames for renaming
        of eref names
    thisModule.ReferencesNamesToList(comp);
  }
}

lazy rule REFRight {
  from
    r : MOFRight!"ecore::EReference",
    l : MOFLeft!"ecore::EPackage",
    c : MOF!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    comp: MOF!"ecore::EReference"(
      -- right mm eReferences need to be changed if pointing to
          a class that has been renamed
      -- but now we just copy them, as the modifications can
          only be done at the end
      name <-  r.name,
      lowerBound <- r.lowerBound,
      upperBound <- r.upperBound,
      eType <- r.eReferenceType,
      containment <- r.containment
    )
  do {
    -- feeling the collection of ereferences
    thisModule.ReferencesToList(comp);
  }
}

-- FEELING COLLECTION RULES

rule ReferencesToList (e : MOF!EReference) {
  do {
    thisModule.referencesFromResultMM <- thisModule.
        referencesFromResultMM->append(e);
  }
}

rule ReferencesNamesToList (e : MOF!EReference) {
  do {
    thisModule.referencesNamesFromLMM <- thisModule.
        referencesNamesFromLMM->append(e.name);
  }
}

rule classesFromRMM2List (e : MOFRigth!EClass) {
  do {
    thisModule.classesFromRMM <- thisModule.classesFromRMM->
        append(e.name);
  }
}

rule AttributesToList (e : MOF!EAttributes) {
  do {
    thisModule.attributesFromResultMM <- thisModule.
        attributesFromResultMM->append(e);
  }
}

rule classesThatHaveSuperTypes2List (e : MOFRigth!EClass) {
  do {
    thisModule.classesThatHaveSuperTypes <- thisModule.
        classesThatHaveSuperTypes->append(e);
  }
}

-- EXECUTION OF DELAYED ACTIONS

helper def: listOfMergedClasses2 : OrderedSet(MOFRight!EClass) =
     OrderedSet{};
helper def: listOfMergedClassesAttributes : OrderedSet(MOFRight!
    EAttribute) = OrderedSet{};

endpoint rule EndRule() {
  do {

    -- rule for getting the merged classes in order to be able
        to add eAttributes
    thisModule.listOfMergedClasses2  <- thisModule.
        listOfMergedClasses2.union(
    (thisModule.thePackageFromRightMM.eClassifiers->select(e |
        thisModule.thePackageFromLeftMM.getAllClassesNames().
        includes(e.name))->iterate( c ; elements : OrderedSet(
        MOFRight!EClass) =
      OrderedSet{} | elements.append(c))));

    -- now we can iterate on each merged class from the right
        meta model
    for(dta in thisModule.listOfMergedClasses2) {
      -- we use this rule for getting a list with all the
          attributes of the merged class
      -- that we should add to the class in the resulting meta
          model
      thisModule.listOfMergedClassesAttributes <- thisModule.
          listOfMergedClassesAttributes.union(
        (dta.eAttributes->select(e | (MOF!EClass.allInstances()
            ->any(e | e.name = dta.name)).eAttributes->collect(e
            | e.name).excludes(e.name) )->iterate(c; elements :
              OrderedSet(MOFRight!EAttribute) =
            OrderedSet{} | elements.append(c))));
      -- then we need to iterate on those attributes
      for(dtb in thisModule.listOfMergedClassesAttributes) {
        -- and add them to the eStructuralFeatures list of the
              corresponding resulting class
        (MOF!EClass.allInstances()->any(e | e.name = dta.name)).
              eStructuralFeatures<-dtb;
      }
    }

    -- renaming and modifying eAttributes
    for(dta in thisModule.attributesFromResultMM) {
      dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eType.name))
              then
                (
                (MOF!EClass.allInstances()->any(e | e.name = dta.
                    eType.name))
                )
              else
                (
                (MOF!EClass.allInstances()->any(e | e.name = (dta.
                    eType.name + '_RMM')))
                )
            endif;
    }
    -- for the inheritances relation adaptation
    for(dta in thisModule.classesThatHaveSuperTypes) {
      for(dtb in dta.eSuperTypes) {
        MOF!EClass.allInstances()->any(e | e.name = dta.name or
              e.name = (dta.name + '_RMM')).eSuperTypes <-
          if (thisModule.classesFromRMM.includes(dtb.name))
              then
                (
                (MOF!EClass.allInstances()->any(e | e.name = dtb.
                    name))
                )
              else
                (
                (MOF!EClass.allInstances()->any(e | e.name = (dtb.
                    name + '_RMM')))
                )
            endif;
      }
    }
    -- renaming and modifying eReferences
    for(dta in thisModule.referencesFromResultMM) {
      -- we can now rename the eReference name
      dta.name <- if(thisModule.referencesNamesFromLMM.includes(
          dta.name))
                then
                  (dta.name+'_RMM')
                else
                  (dta.name)
              endif;
      -- and modify the binding to the good EClass
      dta.eType <- if (thisModule.classesFromRMM.includes(dta.
          eReferenceType.name))
                then
                  (
                   (MOF!EClass.allInstances()->any(e | e.name =
                        dta.eReferenceType.name))
                  )
                else
                  (
                   (MOF!EClass.allInstances()->any(e | e.name = (
                        dta.eReferenceType.name + '_RMM')))
                  )
              endif;
    }
  }
}
```

A.2.3 Template for Association Operator

Listing A.4. ATL Template Transformation for Association operator

```
-- @atlcompiler atl2006
module association_execute_metamodel_transformation; -- Module
    Template
create metamodelcomposed : MOF from modelLeft : MOFLeft,
    modelRight: MOFRight;

-- HELPING METHODS

helper def: referencesForContainment : OrderedSet(MOF!EReference
    ) =
  OrderedSet {};
```

```
-- INITIALIZATION

rule init {
  from
    lPack : MOFLeft!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    compPack: MOF!"ecore::EPackage"(
      -- setting name of the new epackage
      name <- lPack.name + '_association_' + rPack.name,
      -- setting the eclassifies from left and right meta model
      eClassifiers <- lPack.eClassifiers,
      eClassifiers <- rPack.eClassifiers
      )
}

-- MANAGING ECLASSES

rule CLASSLeft {
  from
    l : MOFLeft!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

rule CLASSRight1 {
  from
    l : MOFRight!"ecore::EClass" (l.name='@@suplierCName@@')
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    ),
    comp2: MOF!"ecore::EReference"(
        -- left mm eReferences just need to be copied in the
             result mm
        name <-  l.name.toLower(),
        containment <- false,
        lowerBound <- @@suplierLBound@@,
        upperBound <- @@suplierUBound@@,
        eType <- comp
      )
  do {
    thisModule.ReferencesForContainmentToList(comp2);
  }
}

rule CLASSRight2 {
  from
    l : MOFRight!"ecore::EClass"  (l.name<>'@@suplierCName@@')
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
           in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

-- MANAGING EATTRIBUTES

rule ATRLeft {
  from
    l : MOFLeft!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- eattributes from left mm just need to be put in the
           result mm
      name <-  l.name,
      eType <- l.eType
    )
}

rule ATRRight {
  from
    l : MOFRight!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
        -- eattributes from left mm just need to be put in the
           result mm
        name <-  l.name,
        eType <- l.eType
      )
}

-- MANAGING EREFRENCES

rule REFLeft {
  from
      l : MOFLeft!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
        -- left mm eReferences just need to be copied in the
           result mm
        name <-  l.name,
        upperBound <- l.upperBound,
        eType <- l.eType,
        containment <- l.containment
      )
}

rule REFRight {
  from
      l : MOFRight!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
        -- left mm eReferences just need to be copied in the
           result mm
        name <-  l.name,
        upperBound <- l.upperBound,
        eType <- l.eType,
        containment <- l.containment
      )
}

-- FEELING COLLECTION RULES

rule ReferencesForContainmentToList (e : MOF!EReference) {
  do {
    thisModule.referencesForContainment <- thisModule.
        referencesForContainment->append(e);
  }
}
```

```

-- EXECUTION OF DELAYED ACTIONS

endpoint rule EndRule() {
  do {
    for(dta in thisModule.referencesForContainment) {
      dta.containment <- false;
      (MOF!EClass.allInstances()->any(e | e.name = '
          @@clientCName@@')).eStructuralFeatures <- dta;
    }
  }
}
```

A.2.4 Template for Aggregation Operator

Listing A.5. ATL Template Transformation for Aggregation operator

```
-- @atlcompiler atl2006
module containment_execute_metamodel_transformation; -- Module
    Template
create metamodelcomposed : MOF from modelLeft : MOFLeft,
    modelRight: MOFRight;

-- HELPING METHODS

helper def: referencesForContainment : OrderedSet(MOF!EReference
    ) =
  OrderedSet {};

-- INITIALIZATION

rule init {
  from
    lPack : MOFLeft!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    compPack: MOF!"ecore::EPackage"(
      -- setting name of the new epackage
      name <- lPack.name + '_containment_' + rPack.name,
      -- setting the eclassifies from left and right meta model
      eClassifiers <- lPack.eClassifiers,
      eClassifiers <- rPack.eClassifiers
      )
}
```

```
-- MANAGING ECLASSES

rule CLASSLeft {
  from
    l : MOFLeft!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <- l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

rule CLASSRight1 {
  from
    l : MOFRight!"ecore::EClass" (@@containedList1@@)
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <- l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    ),
    comp2: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
          result mm
      name <- l.name.toLower(),
      containment <- true,
      lowerBound <- @@lBoundList@@,
      upperBound <- @@uBoundList@@,
      eType <- comp
    )
  do {
    thisModule.ReferencesForContainmentToList(comp2);
  }
}

rule CLASSRight2 {
  from
    l : MOFRight!"ecore::EClass" (@@containedList2@@)
  to
```

```
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
         in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

-- MANAGING EATTRIBUTES

rule ATRLeft {
  from
    l : MOFLeft!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- eattributes from left mm just need to be put in the
         result mm
      name <-  l.name,
      eType <- l.eType
    )
}

rule ATRRight {
  from
    l : MOFRight!"ecore::EAttribute"
  to
    comp: MOF!"ecore::EAttribute"(
      -- eattributes from left mm just need to be put in the
         result mm
      name <-  l.name,
      eType <- l.eType
    )
}

-- MANAGING EREFRENCES

rule REFLeft {
  from
    l : MOFLeft!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
         result mm
      name <-  l.name,
        upperBound <- l.upperBound,
        eType <- l.eType,
        containment <- l.containment
      )
}

rule REFRight {
  from
      l : MOFRight!"ecore::EReference"
  to
     comp: MOF!"ecore::EReference"(
       -- left mm eReferences just need to be copied in the
           result mm
        name <-  l.name,
        upperBound <- l.upperBound,
        eType <- l.eType,
        containment <- l.containment
      )
}

-- FEELING COLLECTION RULES

rule ReferencesForContainmentToList (e : MOF!EReference) {
  do {
     thisModule.referencesForContainment <- thisModule.
         referencesForContainment->append(e);
  }
}

-- EXECUTION OF DELAYED ACTIONS

endpoint rule EndRule() {
  do {
     for(dta in thisModule.referencesForContainment) {
       (MOF!EClass.allInstances()->any(e | e.name = '
           @@containerCName@@')).eStructuralFeatures <- dta;
     }
  }
}
```

A.2.5 Template for Inherit Operator

Listing A.6. ATL Template Transformation for Inherit operator

```
-- @atlcompiler atl2006
module inherit_execute_metamodel_transformation; -- Module
    Template
create metamodelcomposed : MOF from modelLeft : MOFLeft,
    modelRight: MOFRight;

-- INITIALIZATION

rule init {
  from
    lPack : MOFLeft!"ecore::EPackage",
    rPack : MOFRight!"ecore::EPackage"
  to
    compPack: MOF!"ecore::EPackage"(
      -- setting name of the new epackage
      name <- lPack.name + '_inherit_' + rPack.name,
      -- setting the eclassifies from left and right meta model
      eClassifiers <- lPack.eClassifiers ,
      eClassifiers <- rPack.eClassifiers
      )
}

-- MANAGING ECLASSES

rule CLASSLeft {
  from
    l : MOFLeft!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
      eSuperTypes <- l.eSuperTypes,
      eStructuralFeatures <- l.eStructuralFeatures
    )
}

rule CLASSRight {
  from
    l : MOFRight!"ecore::EClass"
  to
    comp: MOF!"ecore::EClass"(
      -- for left eClasses we just need to put them as they are
          in the result mm
      name <-  l.name,
        eSuperTypes <- l.eSuperTypes,
        eStructuralFeatures <- l.eStructuralFeatures
      )
   do {
      if(comp.name='@@specializationClass@@') {
        comp.eSuperTypes <- (MOF!EClass.allInstances()->any(e | e.
            name = '@@specializedClass@@'));
      }
      --thisModule.ReferencesForContainmentToList(comp2);
   }
}

-- MANAGING EATTRIBUTES

rule ATRLeft {
   from
      l : MOFLeft!"ecore::EAttribute"
   to
      comp: MOF!"ecore::EAttribute"(
        -- eattributes from left mm just need to be put in the
            result mm
        name <- l.name,
        eType <- l.eType
      )
}

rule ATRRight {
   from
      l : MOFRight!"ecore::EAttribute"
   to
      comp: MOF!"ecore::EAttribute"(
        -- eattributes from left mm just need to be put in the
            result mm
        name <- l.name,
        eType <- l.eType
      )
}

-- MANAGING EREFRENCES

rule REFLeft {
   from
      l : MOFLeft!"ecore::EReference"
   to
      comp: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
          result mm
      name <-  l.name,
      upperBound <- l.upperBound,
      eType <- l.eType,
      containment <- l.containment
    )
}

rule REFRight {
  from
    l : MOFRight!"ecore::EReference"
  to
    comp: MOF!"ecore::EReference"(
      -- left mm eReferences just need to be copied in the
          result mm
      name <-  l.name,
      upperBound <- l.upperBound,
      eType <- l.eType,
      containment <- l.containment
    )
}
```

A.3 ATL Transformation Rules Generation Templates

You can find here the code listings for the ATL Transformation Rules Generation Templates for each meta model composition operator.

A.3.1 Template for Association, Containment and Inherit Operators

Listing A.7. ATL Transformation Rules Templates for Association, Containment and Inherit Operator

```
-- @atlcompiler atl2006
module generate_model_transformation_rules; -- Module Template
create trRules : TrRulesMM from mmLeft : MOF, mmRight : MOF;

-- method for getting a collection of all the class names from
    the left package
```

```
helper context MOF!EClass def : getRulesExpForAttributes () :
    OrderedSet(MOF!EAttribute) =
  self.eAttributes->iterate( c ; elements : OrderedSet(MOF!
      EAttribute) =
    OrderedSet{} | elements.append(c.debug('attribute'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOF!EClass def : getRulesExpForReferences () :
    OrderedSet(MOF!EReference) =
  self.eReferences->iterate( c ; elements : OrderedSet(MOF!
      EReference) =
    OrderedSet{} | elements.append(c.debug('reference'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOF!EPackage def : getRulesExpForClasses () :
    OrderedSet(MOF!EClass) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(MOF!
      EClass) =
    OrderedSet{} | elements.append(c.debug('class'))
    )
  ;

rule init1 {
  from
    pack : MOF!"ecore::EPackage" (pack.name='@@leftPackageName@@
        ')
  to
    root: TrRulesMM!Root (
      left <- pack.eClassifiers->select(a | a.name='
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesA(e,pack)),
      left <- pack.eClassifiers->select(a | a.name<>'
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesB(e,pack))

    )
  do {
    pack.name.debug('T1');
  }
```

```
}


rule init2 {
  from
    pack : MOF!"ecore::EPackage" (pack.name='
        @@rightPackageName@@')
  to
    root: TrRulesMM!Root (
      right <- pack.eClassifiers->select(a | a.name='
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesA(e,pack)),
      right <- pack.eClassifiers->select(a | a.name<>'
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesB(e,pack))

    )
  do {
    pack.name.debug('T2');
  }
}

lazy rule TrRule4LeftEClassesA {
  from
    currentEClass : MOF!"ecore::EClass",
    currentPack : MOF!"ecore::EPackage"
  to
    ruleExpression: TrRulesMM!TrRule (
      ruleExp <- 'rule L_'+currentEClass.name+' { '
                 +'from pack : MMLeft!"'+currentEClass.name+'"'
                 +' to newlPack: ComposedMM!"'+currentEClass.name+'
                     "('
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                    ';
              }
            }
            -- for references
            currentEClass.getRulesExpForReferences().size().debug('
                sizeOfReferences');
            if(currentEClass.getRulesExpForAttributes().size()>0 and
                 currentEClass.getRulesExpForReferences().size()>0) {
              ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
            }
            for(dta in currentEClass.getRulesExpForReferences()) {
              ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
                  dta.name+' <- pack.'+dta.name+'->collect(e |
                  thisModule.L_'+dta.eReferenceType.name+' (e) ) ';
              if (currentEClass.getRulesExpForReferences().indexOf(
                  dta)<>currentEClass.getRulesExpForReferences().size
                  ()) {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                    ';
              }
            }
            -- for closing the rule
            ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
      }
}

lazy rule TrRule4LeftEClassesB {
   from
      currentEClass : MOF!"ecore::EClass",
      currentPack : MOF!"ecore::EPackage"
   to
      ruleExpression: TrRulesMM!TrRule (

         ruleExp <- if(
              currentPack.eClassifiers
                ->any(e | e.name = '@@leftRootCName@@').eReferences
                  ->select(f | f.eType=currentEClass ).size()>0)
                then (
                  'lazy rule L_'+currentEClass.name+' { '
                  +'from pack : MMLeft!"'+currentEClass.name+'"'
                  +' to newlPack: ComposedMM!"'+currentEClass.name+'
                      "('
                )
                else (
            'rule L_'+currentEClass.name+' { '
            +'from pack : MMLeft!"'+currentEClass.name+'"'
            +' to newlPack: ComposedMM!"'+currentEClass.name+'
                "('
          )
        endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
            currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}
```

```
lazy rule TrRule4RightEClassesA {
  from
    currentEClass : MOF!"ecore::EClass",
    currentPack : MOF!"ecore::EPackage"
  to
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <- 'rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!"'+currentEClass.name+'"'
              +' to newrPack: ComposedMM!"'+currentEClass.name+'
                  "('
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
             currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name+'->collect(e |
              thisModule.R_'+dta.eReferenceType.name+' (e) )';
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4RightEClassesB {
  from
    currentEClass : MOF!"ecore::EClass",
    currentPack : MOF!"ecore::EPackage"
  to
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <- if(
          currentPack.eClassifiers
            ->any(e | e.name = '@@rightRootCName@@').eReferences
              ->select(f | f.eType=currentEClass ).size()>0)
            then (
              'lazy rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!"'+currentEClass.name+'"'
              +' to newrPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
            else (
              'rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!"'+currentEClass.name+'"'
              +' to newrPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
          endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
              currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}
```

A.3.2 Template for Union Operator

Listing A.8. ATL Transformation Rules Templates for Union Operator

```
-- @atlcompiler atl2006
module union_generate_model_transformation_rules; -- Module
    Template
create trRules : TrRulesMM from mmLeft : MOFL, mmRight : MOFR;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EClass def : getRulesExpForAttributes () :
    OrderedSet(MOFL!EAttribute) =
  self.eAttributes->iterate( c ; elements : OrderedSet(MOFL!
      EAttribute) =
    OrderedSet{} | elements.append(c.debug('attribute'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
```

```
helper context MOFL!EClass def : getRulesExpForReferences () :
    OrderedSet(MOFL!EReference) =
  self.eReferences->iterate( c ; elements : OrderedSet(MOFL!
      EReference) =
    OrderedSet{} | elements.append(c.debug('reference'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EPackage def : getRulesExpForClasses () :
    OrderedSet(MOFL!EClass) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(MOFL!
      EClass) =
    OrderedSet{} | elements.append(c.debug('class'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EPackage def : getAllClassesNames () :
    OrderedSet(String) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(String)
      =
    OrderedSet{} | elements.append(c.name)
    )
  ;

helper def: thePackageFromLeftMM : MOFL!EPackage = MOFL!EPackage
    ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFR!EClass def : isInLeft () : Boolean =
  if ((MOFL!EPackage.eClassifiers->select(e | true)).includes(
      self)=true)
    then (true)
    else (false)
  endif
  ;

rule init1 {
  from
    pack : MOFL!"ecore::EPackage" (pack.name='
        @@leftPackageName@@')
  to
    root: TrRulesMM!Root (
      left <- pack.eClassifiers->select(a | a.name='
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesA(e,pack)),
      left <- pack.eClassifiers->select(a | a.name<>'
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesB(e,pack))
    )
  do {
    pack.name.debug('T1');
    thisModule.thePackageFromLeftMM<-pack;
  }
}


rule init2 {
  from
    pack : MOFR!"ecore::EPackage" (pack.name='
        @@rightPackageName@@')
  to
    root: TrRulesMM!Root (
      right <- pack.eClassifiers->select(a | a.name='
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesA(e,pack)),
      right <- pack.eClassifiers->select(a | a.name<>'
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesB(e,pack))
    )
  do {
    pack.name.debug('T2');
  }
}

lazy rule TrRule4LeftEClassesA {
  from
    currentEClass : MOFL!"ecore::EClass",
    currentPack : MOFL!"ecore::EPackage"
  to
    ruleExpression: TrRulesMM!TrRule (
      ruleExp <- 'rule L_'+currentEClass.name+' { '
                +'from pack : MMLeft!"'+currentEClass.name+'"'
                +' to newlPack: ComposedMM!"'+currentEClass.name+'
                    "('
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
             currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name+'->collect(e |
              thisModule.L_'+dta.eReferenceType.name+' (e) ) ';
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4LeftEClassesB {
  from
    currentEClass : MOFL!"ecore::EClass",
    currentPack : MOFL!"ecore::EPackage"
  to
    ruleExpression: TrRulesMM!TrRule (

      ruleExp <- if(
          currentPack.eClassifiers
            ->any(e | e.name = '@@leftRootCName@@').eReferences
              ->select(f | f.eType=currentEClass ).size()>0)
            then (
              'lazy rule L_'+currentEClass.name+' { '
              +'from pack : MMLeft!"'+currentEClass.name+'"'
              +' to newlPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
            else (
              'rule L_'+currentEClass.name+' { '
              +'from pack : MMLeft!"'+currentEClass.name+'"'
              +' to newlPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
          endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
             currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4RightEClassesA {
  from
    currentEClass : MOFR!"ecore::EClass",
    currentPack : MOFR!"ecore::EPackage"
  to
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <- if(thisModule.thePackageFromLeftMM.
          getAllClassesNames().includes(currentEClass.name))
            then (
              -- if we need to rename
              'rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!"'+currentEClass.name+'"'
              +' to newrPack: ComposedMM!"'+currentEClass.name+'
                  _RMM"('
            )
            else (
              -- no renaming
              'rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!"'+currentEClass.name+'"'
              +' to newrPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
            endif
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
            currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          if(thisModule.thePackageFromLeftMM.getAllClassesNames
              ().includes(currentEClass.name)) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+'_RMM <- pack.'+dta.name+'->collect(e |
                  thisModule.R_'+dta.eReferenceType.name+' (e) ) '
                ;
          }
          else {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+' <- pack.'+dta.name+'->collect(e |
                thisModule.R_'+dta.eReferenceType.name+' (e) ) ';
          }

          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4RightEClassesB {
  from
    currentEClass : MOFR!"ecore::EClass",
    currentPack : MOFR!"ecore::EPackage"
  to
```

```
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <- if(thisModule.thePackageFromLeftMM.
          getAllClassesNames().includes(currentEClass.name))
            then (
              -- if we need to rename
              if(
              currentPack.eClassifiers
                ->any(e | e.name = '@@rightRootCName@@').
                    eReferences
                  ->select(f | f.eType=currentEClass ).size()>0)
                then (
                  'lazy rule R_'+currentEClass.name+' { '
                  +'from pack : MMRight!"'+currentEClass.name+'"
                    '
                  +' to newrPack: ComposedMM!"'+currentEClass.
                      name+'_RMM"( '
                )
                else (
                  'rule R_'+currentEClass.name+' { '
                  +'from pack : MMRight!"'+currentEClass.name+'"
                    '
                  +' to newrPack: ComposedMM!"'+currentEClass.
                      name+'_RMM"( '
                )
              endif
            )
            else (
              -- no renaming
              if(
              currentPack.eClassifiers
                ->any(e | e.name = '@@rightRootCName@@').
                    eReferences
                  ->select(f | f.eType=currentEClass ).size()>0)
                then (
                  'lazy rule R_'+currentEClass.name+' { '
                  +'from pack : MMRight!"'+currentEClass.name+'"
                    '
                  +' to newrPack: ComposedMM!"'+currentEClass.
                      name+'"( '
                )
                else (
                  'rule R_'+currentEClass.name+' { '
                  +'from pack : MMRight!"'+currentEClass.name+'"
                    '
                  +' to newrPack: ComposedMM!" '+currentEClass.
                      name+'"('
                )
              endif
            )
            endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
                dta.name+' <- pack.'+dta.name;
            if (currentEClass.getRulesExpForAttributes().indexOf(
                dta)<>currentEClass.getRulesExpForAttributes().size
                ()) {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                    ';
            }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
             currentEClass.getRulesExpForReferences().size()>0) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
            if(thisModule.thePackageFromLeftMM.getAllClassesNames
                ().includes(currentEClass.name)) {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                    +dta.name+'_RMM <- pack.'+dta.name;
            }
            else {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                    +dta.name+' <- pack.'+dta.name;
            }

            if (currentEClass.getRulesExpForReferences().indexOf(
                dta)<>currentEClass.getRulesExpForReferences().size
                ()) {
```

```
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}
```

A.3.3 Template for Merge Operator

Listing A.9. ATL Transformation Rules Templates for Merge Operator

```
-- @atlcompiler atl2006
module merge_generate_model_transformation_rules; -- Module
    Template
create trRules : TrRulesMM from mmLeft : MOFL, mmRight : MOFR;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EClass def : getRulesExpForAttributes () :
    OrderedSet(MOFL!EAttribute) =
  self.eAttributes->iterate( c ; elements : OrderedSet(MOFL!
      EAttribute) =
   OrderedSet{} | elements.append(c.debug('attribute'))
   )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EClass def : getRulesExpForReferences () :
    OrderedSet(MOFL!EReference) =
  self.eReferences->iterate( c ; elements : OrderedSet(MOFL!
      EReference) =
   OrderedSet{} | elements.append(c.debug('reference'))
   )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EPackage def : getRulesExpForClasses () :
    OrderedSet(MOFL!EClass) =
  self.eClassifiers->iterate( c ; elements : OrderedSet(MOFL!
      EClass) =
    OrderedSet{} | elements.append(c.debug('class'))
    )
  ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFL!EPackage def : getAllClassesNames () :
    OrderedSet(String) =
  (self.eClassifiers->collect(e | e.eReferences)->flatten()->
      asSet())->iterate( c ; elements : OrderedSet(String) =
    OrderedSet{} | elements.append(c.name)
    )
  ;

helper def: thePackageFromLeftMM : MOFL!EPackage = MOFL!EPackage
    ;

-- method for getting a collection of all the class names from
    the left package
helper context MOFR!EClass def : isInLeft () : Boolean =
  if ((MOFL!EPackage.eClassifiers->select(e | true)).includes(
      self)=true)
    then (true)
    else (false)
  endif
  ;


rule init1 {
  from
    pack : MOFL!"ecore::EPackage" (pack.name='
        @@leftPackageName@@')
  to
    root: TrRulesMM!Root (
      left <- pack.eClassifiers->select(a | a.name='
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesA(e,pack)),
      left <- pack.eClassifiers->select(a | a.name<>'
          @@leftRootCName@@')->collect(e | thisModule.
          TrRule4LeftEClassesB(e,pack))
    )
  do {
    pack.name.debug('T1');
    thisModule.thePackageFromLeftMM<-pack;
  }
```

```
}

rule init2 {
  from
    pack : MOFR!"ecore::EPackage" (pack.name='
        @@rightPackageName@@')
  to
    root: TrRulesMM!Root (
      right <- pack.eClassifiers->select(a | a.name='
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesA(e,pack)),
      right <- pack.eClassifiers->select(a | a.name<>'
          @@rightRootCName@@')->collect(e | thisModule.
          TrRule4RightEClassesB(e,pack))
    )
  do {
    pack.name.debug('T2');
  }
}

lazy rule TrRule4LeftEClassesA {
  from
    currentEClass : MOFL!"ecore::EClass",
    currentPack : MOFL!"ecore::EPackage"
  to
    ruleExpression: TrRulesMM!TrRule (
      ruleExp <- 'rule L_'+currentEClass.name+' { '
              +'from pack : MMLeft!"'+currentEClass.name+'"'
              +' to newlPack: ComposedMM!"'+currentEClass.name+'
                  "('
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
                }
            }
            -- for references
            currentEClass.getRulesExpForReferences().size().debug('
                sizeOfReferences');
            if(currentEClass.getRulesExpForAttributes().size()>0 and
                 currentEClass.getRulesExpForReferences().size()>0) {
              ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
            }
            for(dta in currentEClass.getRulesExpForReferences()) {
              ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
                  dta.name+' <- pack.'+dta.name+'->collect(e |
                  thisModule.L_'+dta.eReferenceType.name+' (e) ) ';
              if (currentEClass.getRulesExpForReferences().indexOf(
                  dta)<>currentEClass.getRulesExpForReferences().size
                  ()) {
                ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                    ';
              }
            }
            -- for closing the rule
            ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
        }
}

lazy rule TrRule4LeftEClassesB {
  from
    currentEClass : MOFL!"ecore::EClass",
    currentPack : MOFL!"ecore::EPackage"
  to
    ruleExpression: TrRulesMM!TrRule (

      ruleExp <- if(
          currentPack.eClassifiers
            ->any(e | e.name = '@@leftRootCName@@').eReferences
              ->select(f | f.eType=currentEClass ).size()>0)
            then (
              'lazy rule L_'+currentEClass.name+' { '
              +'from pack : MMLeft!"'+currentEClass.name+'"'
              +' to newlPack: ComposedMM!"'+currentEClass.name+'
                  "('
            )
            else (
              'rule L_'+currentEClass.name+' { '
              +'from pack : MMLeft!"'+currentEClass.name+'"'
            +' to newlPack: ComposedMM!" '+currentEClass.name+'
                "('
          )
        endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
              currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4RightEClassesA {
  from
```

```
    currentEClass : MOFR!"ecore::EClass",
    currentPack : MOFR!"ecore::EPackage"
  to
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <-  'rule R_'+currentEClass.name+' { '
              +'from pack : MMRight!" '+currentEClass.name+'" '
              +' to newrPack: ComposedMM!" '+currentEClass.name+'
                  "( '
    )
    do {
        -- CASE OF LAZY RULE 4 PACKAGE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
            currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          if(thisModule.thePackageFromLeftMM.getAllClassesNames
              ().includes(dta.name)) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+'_RMM <- pack.'+dta.name+'_RMM->collect
                (e | thisModule.R_'+dta.eReferenceType.name+' (e)
                 ) ';
          }
          else {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+' <- pack.'+dta.name+'->collect(e |
                thisModule.R_'+dta.eReferenceType.name+' (e) ) ';
          }
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}

lazy rule TrRule4RightEClassesB {
  from
    currentEClass : MOFR!"ecore::EClass",
    currentPack : MOFR!"ecore::EPackage"
  to
    ruleExpression : TrRulesMM!TrRule (
      ruleExp <- if(
                currentPack.eClassifiers
                  ->any(e | e.name = '@@rightRootCName@@').
                      eReferences
                    ->select(f | f.eType=currentEClass ).size()>0)
                  then (
                    'lazy rule R_'+currentEClass.name+' { '
                    +'from pack : MMRight!"'+currentEClass.name+'"
                        '
                    +' to newrPack: ComposedMM!"'+currentEClass.
                        name+'"('
                  )
                  else (
                    'rule R_'+currentEClass.name+' { '
                    +'from pack : MMRight!"'+currentEClass.name+'"
                        '
                    +' to newrPack: ComposedMM!"'+currentEClass.
                        name+'"('
                  )
                endif
    )
    do {
        -- DEFAULT CASE

        -- for attributes
        currentEClass.getRulesExpForAttributes().size().debug('
            sizeOfAttributes');
        for(dta in currentEClass.getRulesExpForAttributes()) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+' '+
              dta.name+' <- pack.'+dta.name;
          if (currentEClass.getRulesExpForAttributes().indexOf(
              dta)<>currentEClass.getRulesExpForAttributes().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for references
        currentEClass.getRulesExpForReferences().size().debug('
            sizeOfReferences');
        if(currentEClass.getRulesExpForAttributes().size()>0 and
             currentEClass.getRulesExpForReferences().size()>0) {
          ruleExpression.ruleExp <- ruleExpression.ruleExp+', ';
        }
        for(dta in currentEClass.getRulesExpForReferences()) {
          if(thisModule.thePackageFromLeftMM.getAllClassesNames
              ().includes(dta.name)) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+'_RMM <- pack.'+dta.name+'_RMM';
          }
          else {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+' '
                +dta.name+' <- pack.'+dta.name;
          }
          if (currentEClass.getRulesExpForReferences().indexOf(
              dta)<>currentEClass.getRulesExpForReferences().size
              ()) {
            ruleExpression.ruleExp <- ruleExpression.ruleExp+',
                ';
          }
        }
        -- for closing the rule
        ruleExpression.ruleExp <- ruleExpression.ruleExp+') }';
    }
}
```

A.4 ATL Transformation for Model Composition Template

You can find here the code listing for the Transformation for Model Composition Template used by all meta model composition operators.

A.4.1 Template for all Operators

Listing A.10. ATL Transformation for Model Composition Template for all Operators

```
-- @atlcompiler atl2006
module execute_model_transformation; -- Module Template
create newModel : ComposedMM from modelLeft : MMLeft, modelRight
      : MMRight;

-- INITIALIZATION
-- simple copy of the elements of left and right models
-- to the new model conforming to composed meta model

@@modelTransformationRules@@

-- EXECUTION OF DELAYED ACTIONS
-- rules depending on the operator

-- rules for operator
endpoint rule EndRule() {
  do {
    -- rule depending on operator
    @@operatorRules@@
  }
}
```

Appendix B

Acronyms

ADT Abstract (Algebraic) Data Type

API Application Programming Interface

ATL ATLAS Transformation Language

CIM Computation Independent Model

COOPN Concurrent Object Oriented Petri Nets

DOM Document Object Model

DC Domain Concept

EMF Eclipse Modeling Framework

EPL Eclipse Public License

FST Formal Specification Techniques

GMT Generic Mapping Tools

GME Generic Modeling Environment

IDE Integrated Development Environment

INRIA Institut National de Recherche en Informatique et en Automatique (France)

IST Informal Structured Techniques

MDA Model-Driven Architecture

MOF Meta Object Facilities

MTV Model Transformation for Verification

OCL Object Constraint Language

OMG Object Management Group

OO Object Oriented

PDAs Personal Digital Assistants

PIM Platform Independent Model

PSM Platform Specific Model

PN Petri Nets

QVT Query/View/Transformation

RFP Request For Proposal

SMV Software Modeling and Verification

SUT System Under Test

UML Unified Modeling Language

URL Uniform Resource Locator

W3C World Wide Web Consortium

WWW World Wide Web

XML Extensible Markup Language

XMI XML Metadata Interchange

Bibliography

[1] Eclipse. Eclipse Modeling Framework. URL: `http://eclipse.org/emf/`.

[2] Didier Buchs and SMV Group from University of Geneva. Concurrent Object Oriented Petri Nets. URL: `http://smv.unige.ch/tiki-index.php?page=IntroCoopn`.

[3] Stephane Heck. Model transformation and verification: building the basis for a generic tool. Bachelor thesis, Centre Universitaire D'Informatique, Universite de Genève, September 2005.

[4] Software Modeling and Verification. Research Group - University of Geneva. URL: `http://smv.unige.ch/`.

[5] Object Management Group. Mda guide version 1.0.1. Technical report, OMG, June 2003.

[6] Object Management Group. URL: `http://www.omg.org/`.

[7] MetaCase Consulting. Metaedit+. URL: http://www.metacase.com.

[8] Vanderbilt University Institute for Software Integrated Systems. Gme 2000 and metagme 2000. URL: http://www.isis.vanderbilt.edu.

[9] Computer Associates. Paradigm plus. URL: http://ca.com/products.

[10] Honeywell International Inc. Domain modeling environment. URL: http://www.htc.honeywell.com/dome/.

[11] Didier Buchs. Software Engineering course. 2005 - 2006. URL: `http://smv.unige.ch/`.

[12] OMG. Meta-Object Facility Specification. URL: `http://www.omg.org/technology/documents/formal/mof.htm`.

[13] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. the eclipse series. Addison Wesley, 2004.

[14] ATLAS Group (INRIA LINA). ATLAS Transformation Language. URL: `http://www.eclipse.org/m2m/atl/`.

[15] Uni Hambourg. Petri Nets. URL: `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`.

[16] SMV Group from University of Geneva. COOPN Blog. URL: `http://coopn.wordpress.com/`.

[17] Sergio Coelho. A meta model for coopn language. Master thesis, Centre Universitaire D'Informatique, Universite de Genève, 2008.

[18] OMG. MOF QVT Final Adopted Specification. URL: `http://www.omg.org/docs/ptc/07-07-07.pdf`.

[19] Gabor Karsai Akos Ledeczi, Peter Volgyesi. Metamodel Composition in the Generic Modeling Environment. URL: `http://www.metamodelling.com/ECOOP2001/submissions/Ledeczi_Vanderbilt_ECOOP_WS.pdf`.

[20] Keng Siau. *Advanced Topics in Database Research*. IGI Global, 2006.

[21] ATL News Group. URL: `http://www.eclipse.org/newsportal/thread.php?group=eclipse.modeling.m2m`.

[22] ATL User Manual. URL: `http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf`.