Títol: Processador d'expressions OCL en un entorn de modelització conceptual

Volum: 1/1

Alumne: Villegas Niño, Antonio

Director/Ponent: Olivé Ramon, Antoni

Departament: Llenguatges i Sistemes Informàtics

Data: Juny 2008

DADES DEL PROJECTE

Títol del Projecte:	Processador d'expressions OCL en un entorn de modelització conceptual
Nom de l'estudiant:	Villegas Niño, Antonio
Titulació:	Enginyeria en Informàtica
Crèdits:	37,5
Director/Ponent:	Olivé Ramon, Antoni
Departament:	Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Teniente Lopez, Ernest
Vocal:	González Alastrue, José Antonio
Secretari:	Olivé Ramon, Antoni

QUALIFICACIÓ

Qualificació numèrica:

Qualificacio descriptiva:

Data: Juny 2008

To those who supported me,
and specially to my sister and family.

Contents

Definitions

Examples

Figures

Tables

INTRODUCTION 1

1 INTRODUCTION

1.1 ABOUT THE PROJECT

The OCL Expressions Processor in a Conceptual Modeling Environment project is a final career project made by Antonio Villegas, student of computer engineering at the Barcelona School of Informatics, member of the Technical University of Catalonia (UPC).

This project was born thanks to a pleasant collaboration with the Research Group in Conceptual Modeling of Information Systems (GMC) of the Languages and Systems Department of the UPC, which began at the second trimester of 2007. Their people granted to me a research fellowship in order to improve their conceptual modeling tool, called Eina GMC [GMC07], in the way of adding the processing of Object Constraint Language (OCL, [Obj06]) expressions into it.

Now I present you here the work made during that time. The document that you are reading is the report of this final career project.

1.2 MOTIVATIONS

At the beginning of my collaboration with the GMC Research Group, I had several conversations about what would be my contribution to the group's work. At that time, I was studying a subject on compilers in my penultimate year of the career, and Antoni Olivé, as chief of the group, asked me if I could develop an OCL expressions processor, as I explained before. I accepted the proposal and Antoni became my project director.

It was a great opportunity for me to continue learning about compilers in a practical way, applying my recent knowledge to a different problem. During my years at Barcelona School of Informatics, I enjoyed my lessons about both Software Engineering and Compilers and this was the perfect excuse to mix those topics and develop a final career project, which finalizes my studies.

1.3 PURPOSE

First of all I must specify what are our purposes to achieve during both the development phase and the time after the end of it. This point is very important because to know the purposes of a project is the best way to check if a project is correctly made. You only have to verify the completeness degree of your objectives and assure that they pass the quality threshold of the future users.

The main purpose of this project is to develop an OCL expressions processor in order to use it within a conceptual modeling environment. At this section we will explain this in detail.

When we speak about OCL expressions we refer to grammatical constructions belonging to the Object Constraint Language, in its version 2.0. This language, as we will see deeply at next chapters, allows us to describe formally several rules over a conceptual model, commonly written in UML[Obj07].

The processor, which is the case study of this project, should work with OCL2.0 expressions related to a model as input, in order to fragment them in basic parts, to analyze them, and to create instances of such expression according a superstructure called metamodel, established by the specification that defines the OCL2.0 language.

These instances should be stored in order to keep the data along the time, in a standard format that allows users of the processor to exchange their models with other researchers. Moreover, the processor should distinguish between well-formed expressions and bad ones in order to help users to avoid specification errors while they are writing constraints in OCL2.0 over models.

In the same way, we should offer the inverse processing path, i.e., from stored instances to textual constraints that allow us to recover our parsed expressions from the repository in which they are stored.

As a conclusion, when the project will be finished its users should save time and work on making the specification of information systems thanks to the usability of the processor and its functionality.

To summarize, the purposes over the OCL 2.0 expressions processor are the following:

- Convert textual constraints in OCL 2.0 over a model written in UML 2.0 to metamodel instances according the specification of both OCL 2.0 and UML 2.0 modeling languages.

- Convert metamodel instances to textual constraints in OCL 2.0, or in other words, the inverse processing path of our tool.

- Detect possible errors and give feedback to users in order to help them to solve the possible mistakes and obtain well-formed expressions (for apply to the UML models).

- Import/export stored OCL 2.0 expressions from/to a standard interchange format that allow users to keep them safe.

- Delete OCL 2.0 expressions that have been previously processed and stored.

1.4 DEVELOPMENT METHOD

Nowadays, to carry out the task of building a software system from the ground up isn’t worth it because there are a lot of ways to find some pieces of code that you can reuse or adapt to your needs.

Following this philosophy, first of all we studied the existent alternatives of OCL 2.0 tools and their features in order to choose the one that adjust better to our solution in mind. This decision problem has a huge importance, so we decided to dedicate the treatment of it in a full chapter, as we can find at chapter 10.

Once we chose our alternative we proceed to study it in detail, including its features and the respective source code in order to understand them better and to know how we can adapt the alternative to our purpose.

Then, we selected a minimum subset of the Object Constraint Language and develop the complete functionalities of our processor with both the adaptation of our chosen alternative and the features of the conceptual modeling environment tool (in our case, Eina GMC). When we covered this hard challenge, we decided to begin an iterative process.

This process follows the next development cycle:

1. Select a new subset of the OCL 2.0 adding more language structures and elements to the previous one.

2. Implement the behaviour of the processor for the new elements in the subset.

3. Test the processor in order to find errors and fix them.

4. Go to step 1 until the subset becomes the complete OCL 2.0.

This iterative and incremental method allowed us to have a complete release version at every iteration end, with all the functionalities working for the subset of the OCL 2.0 that is available.

This way, we could show to final users how the processor works and change it if necessary since the first release version, which implies a low cost, an easiest changeability and a greater degree of users conformity.

1.5 ABOUT THIS REPORT

The aim of this report is to compile and explain all the different phases of work done during the project development. In order to make it more readable for inexperienced users, all the relevant concepts will be explained here. However, some bibliographical notes can be found at the end of the report for to complete and extend the contents that are included here. The rest of this report is structured as follows.

Chapter 2 introduces the conceptual modeling topic and presents why it is important in software development. The Unified Modeling Language (UML, [Obj07]) is explained here as standard modeling language, and a little introduction to the Model Driven Architecture will note why UML alone is not sufficient for make complete models.

Chapter 3 studies the metamodel concept, showing a small subset of the UML Metamodel and explaining it. Also, the Meta-Object Facility (MOF) is explained according to its relationship with UML. Related to it we found at chapter 4 the XML Metadata Interchange (XMI) as a tool with which we are able to represent and share models and metamodels.

Chapters 5 and 6 introduce the conceptual modeling environment (CME) used to be the basis of our project, and the Object Constraint Language (OCL, [Obj06]), respectively. Some notions about how to use Eina GMC as CME and what kind of expressions are possible to make with OCL version 2.0 will be shown there.

Chapter 7 is related with all explained before and shows a description of the OCL 2.0 Metamodel. In addition, some problems found on it during development that force us to make little changes on it are explained next to its solutions.

Because of our purpose is the development of a software tool similar to a compiler, to know the basis about compilers theory and its structure is mandatory. Chapter 8 will show the notions that should be precondition to understand the following chapters.

Chapter 9 and 10 introduce different existent tools made for similar purposes than ours, and the decision making process to choose the one that approximates in a closer way to our solution in mind, respectively. This decision will be taken thanks to a formal process explained there.

Chapter 11 shows the design pattern called Visitor Pattern. This pattern is very related to the development infrastructure of the OCL tool chosen at chapter 9, so this is why is important to know its purpose and implementation.

Chapter 12 introduces what tools were used during the development phase in two ways. Firstly, we will see some tools for help a programmer in the task of constructing a compiler. And then we will pay attention to some Integrated Development Environments (IDEs) where the programming experience becomes easy.

Chapter 13 shows the grammar used to construct the OCL 2.0 expressions processor and its more remarkable aspects and particularities to note. And chapter 14 denotes the development process used, as we partly introduce before at section 1.4 of this starting chapter.

The OCL 2.0 expressions processor (also known as OCL 2.0 parser) and all its details are introduced deeply at chapter 15. All the conversions and their particularities are exemplified there.

To finish the report, some chapters follow chapter 15. Chapter 16 explains different ways to improve and complete the processor in possible future development extensions over it. Immediately, chapter 17 shows the project planning and chapter 18 indicates an estimation of the economic cost of the project.

Last pages are completed with a glossary, a bibliography where to find interesting links, articles and books to consult, and an index of terms.

I hope that you consider the reading of this final career project report useful and, especially, entertained.

CONCEPTUAL MODELING 2

2 CONCEPTUAL MODELING

2.1 INTRODUCTION

Since human beings are able to write computer programs and to develop software systems, the way of doing this task has totally changed.

First programmers were used to write code directly with huge concatenations of 0s and 1s to specify the instructions that a computer had to execute. Obviously, now someone can thing that this is not the best way but it was a common job during a big period of time. Debugging task had to be a heroic work.

The birth of assembler languages was a big revolution in computer programming. Programmers forgot binary arrays to be used with assembler instructions. This simple change provided by programs that convert assembler instructions into machine code introduced a first abstraction layer. At this time, most people did not think that such change could be become in a revolution because first converters had problems. But when such first compilers achieved a minimum degree of quality, assembler programming was the best way of writing code.

Second revolution arrived when a new abstraction layer was introduced by means of the high-level programming languages. Such languages provide simplicity of use that allows programmers to write more readable and easy to debug code with a diminution of work time.

Nowadays a new computer programming generation is growing. Most of people think that it is a chimera that never will work. But we should remember that this situation is the same that happened with assembler instructions, and every day we are closer to make it reality. We talk about conceptual modeling in the scope of both a model-driven architecture and a model-driven development.

Along this chapter we will introduce this idea in order to show that it is the next step in software development. Furthermore we explain why an OCL 2.0 expressions processor is required to help this new abstraction layer become a reality.

2.2 CONCEPTUAL MODELING

In software engineering, we use the name conceptual modeling for the activity that specifies and describes the general knowledge a software system needs to know.

Conceptual modeling is a necessary activity in the development of a software system, the main purpose of which is to define the conceptual schema of such system.

Every software system owns a conceptual schema although it is not always written in a physical document. The whole team of developers of a software system, including designers, programmers and also users, have a conceptual schema of such system in their minds.

The worst problem here is that every member of this team thinks in a different version of the conceptual schema. Therefore, it is a hard but important task to write the conceptual schema of the software system in order to avoid future problems that will not appear at the beginning, if this job is not done.

It is important to emphasize that the later a problem appears the more expensive the solution is, so to find problems early it is important in order to maintain both the time and cost planning.

There exists a principle named as uncertainty principle of the requirements engineering, which describes that the requirements of a software system cannot be validated as well-chosen requirements until the user gets in touch with such system.

Therefore, in the conceptual modeling activity of a software system is important to involve the user of such system in order to create the conceptual schema with his requirements.

A full view about conceptual modeling in the scope of information systems can be found in [Oli07].

2.3 CONCEPTUAL SCHEMAS

A conceptual schema is the result of the conceptual modeling activity. It provides the description of a domain that consists of objects, relationships, and concepts. Conceptual schemas are similar to ontologies, and the languages in which they are written are called conceptual modeling languages.

Specifically, a conceptual schema describes the things of significance to an information system (entity classes), about which it is inclined to collect information and characteristics of their attributes and associations between pairs of those things of significance (relationships).

Conceptual schema: description of the things of significance (objects, relationships and attributes) of a domain. This describes the semantics of an organization and represents a series of assertions about its nature.

Definition 2.1: Conceptual schema

One such language is the Unified Modeling Language (UML) that we will explain along this chapter and later, showing examples of conceptual modeling using the UML with different concepts in a well-known domain.

2.4 THE UNIFIED MODELING LANGUAGE

A complete explanation about the Unified Modeling Language in its version 2.0 can be found in [RJB04]. Our aim in this section is only to describe how to model common concepts like entity types, relationships and attributes about a concrete domain with the Unified Modeling Language.

2.4.1 ENTITY TYPES

An entity type is a concept whose instances at a given time are identifiable individual objects that are considered to exist in the domain at that time.

Let us imagine that our domain is a company that sells products over the world and wants to develop an information system in order to increase the control over its products in an easiest way. It is obvious to think about products and clients as entity types of such system. Therefore we have to model the knowledge about these concepts, and we choose the UML language to do such task.

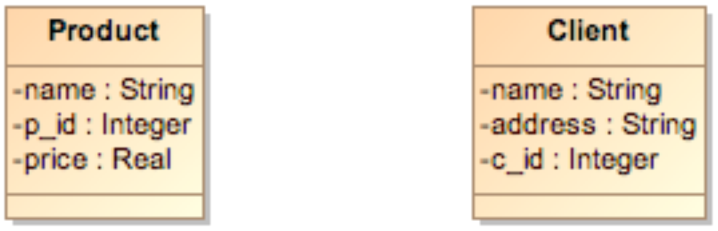


Figure 2.1: Entity types modeled with UML

At Figure 2.1 we can see how to model product and client entity types with the UML. It is easy to do because we only have to draw a rectangle with three divisions. First one is used to place the name, in upper case, of the entity type. Second field of this rectangle is the place for the attributes owned by the entity type. And the last division is for to place operations that belong to such entity.

The whole rectangle is known in UML as UML class. It is important to note that the attributes of a class are the properties or characteristics of the entity type that such class represents.

Furthermore, we have the possibility of model hierarchies of classes. It behaviour can be expressed with a triangular arrow placed from specific classes to the general, as shown in next schema.

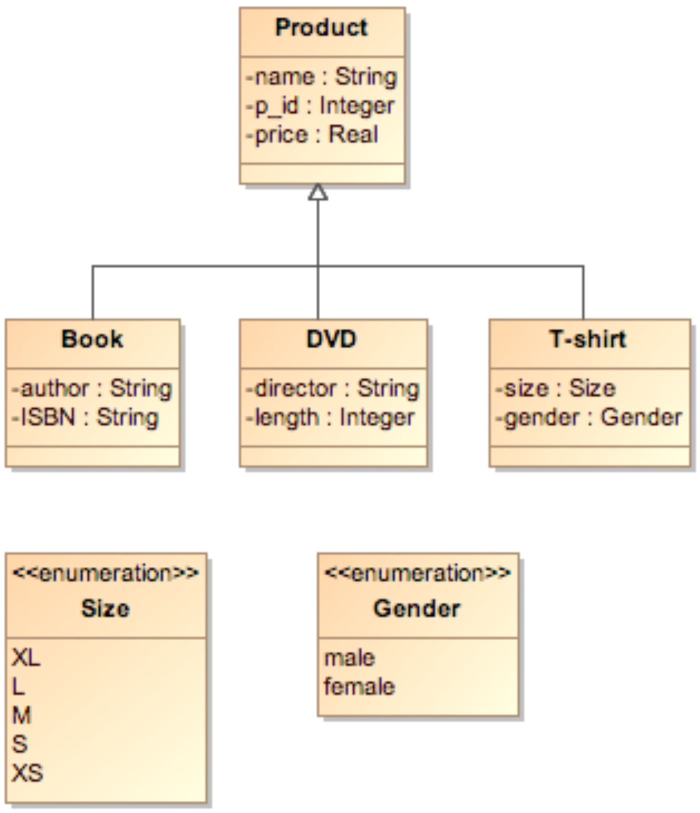


Figure 2.2: Hierarchy and enumerations

It is important to note the possibility of use enumeration of values as types for the attributes in a class. In Figure 2.2 we see that both size and gender attributes owned by T-shirt subclass of product have an enumeration as a type. The way of model and enumeration is similar than modeling a class but indicating with the <<enumeration>> notation that such element is an enumeration of different values.

There exist many Computer Aided Software Engineering (CASE) tools that provide facilities to draw conceptual schemas, so we can use them to model knowledge domains in UML.

2.4.2 Relationships

The Unified Modeling Language also provides the possibility of model relationships between entity types represented as classes.

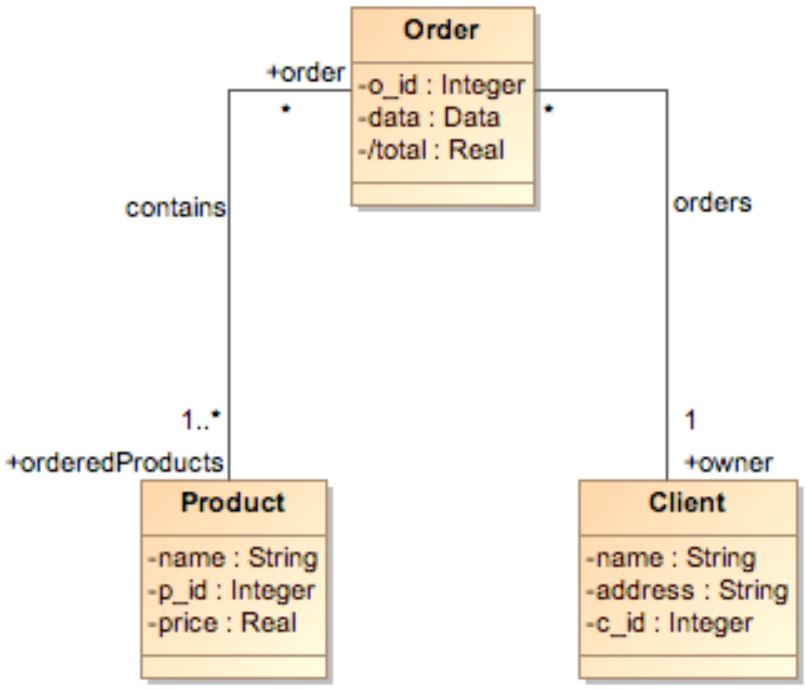


Figure 2.3: Relationships in UML

As we can see in Figure 2.3 binary relationship types in UML are represented by a line between the two members of the association. Each member end can be named with a role name that will be useful to navigate through relationships. Furthermore, such member ends have a multiplicity number representing the number of instances of the class that must have the association.

For example, the *Contains* association has a multiplicity 1..* in its orderedPoducts end. It indicates thatan instance of the Order class has at least one instance of Product owned through this association. The * symbol indicates that the upper value of the number of instances is infinite. If in a multiplicity only appears the * symbol, it is the same as 0..*, i.e., the number of instances may be any positive integer number including 0.

It is important to emphasize that in Order class we can see the attribute *total* with a slash '/' preceding its name. It represents that the attribute is derived, so its value is calculated each time that it is consulted. In this case we can imagine that its value results from the addition of the *price* attributes of the products in the orderedProducts end.

With these simple constructs we are able to model complex information systems using the Unified Modeling Language in order to obtain conceptual schemas representing such systems.

2.5 THE NEED OF TEXTUAL CONSTRAINTS

The information contained in a model has a tendency to be incomplete, informal, ambiguous or imprecise. It is caused by the limitations of the diagrams used, and therefore these limitations are also within the Unified Modeling Language.

For example, in the UML schema shown in Figure 2.3, attribute named total of Order class is derived but this derivation is not specified anywhere. It is impossible to express this derivation rule in the diagram.

```
context Order::total: Real
   derive totalPrice : orderedProducts.price->sum()
```

Example 2.1: Derivation rule

In the previous example we indicate that the evaluation for such attribute is the addition of the price attribute of all products accessible through the navigation by the *contains* relation. We have used the Object Constraint Language (OCL) to express this derivation.

OCL is a precise and unambiguous language that offers benefits over the use of diagrams to specify systems. For instance, different people cannot interpret expressions written in such language differently.

The combination of UML and OCL offers the best of both languages to software developers. To obtain a complete model, both the diagrams and the OCL expressions are essential because without OCL the model becomes underspecified.

It is important to emphasize that OCL expressions alone are not possible because they need UML elements to refer.

We must remember that our aim in this project is to develop a processor for OCL 2.0 expressions. Both UML and OCL are computable with different tools in order to achieve a new abstraction layer in computer programming. This new programming method is the case study of the next section.

2.6 MODEL DRIVEN ARCHITECTURE

Model-driven architecture (MDA) is a software design approach launched by the Object Management Group (OMG, [MDAw]) in 2001. It is becoming an important aspect of software development.

The keystones in MDA are the models. The aim of MDA is that a PIM (high-level model) can be transformed into more than one PSM (low-level model). Therefore, to develop a software system we only have to design its conceptual schema with all constraints using UML and OCL respectively.

This is a new level of abstraction because programmers will change programming languages for modeling languages.

2.6.1 MDA PROCESS

The MDA process is divided into three steps:

1. Build a Platform Independent Model (PIM), that is, a conceptual model of our desired system, which is independent of any implementation technology.

2. Transform the PIM into one or more Platform Specific Models (PSMs) that are based on elements and concepts of the implementation in a specific technology.

3. Transform such PSMs into code.

Therefore, first step is to model the system with a high-level technology-independent language, as for example the combination of UML and OCL. Then we convert such models with other ones that contain elements of the implementation technology chosen. For instance, we could convert our model into a Java model, where UML classes are transformed into Java classes and so on.

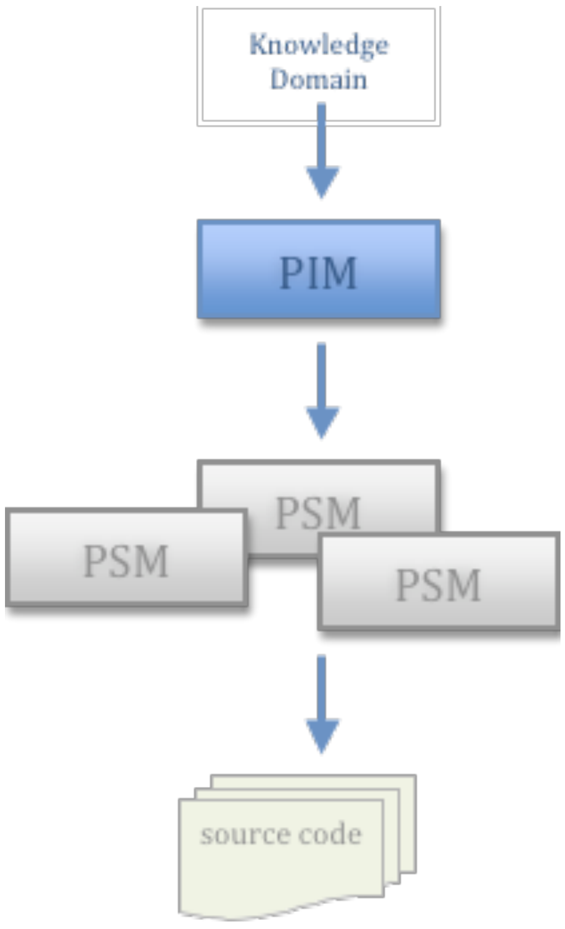


Figure 2.4: Model-Driven Architecture

Finally, last step is to transform the Java model in our case into code. This conversion process is easier that the one between PIM to PSMs.

2.6.2 A CLOSER REALITY

Most of software developers do not believe that MDA may change the future of software development. In my opinion, MDA is still in development because to achieve a MDA tool without problems need a big effort.

Nevertheless, we think that it will become the next step in how to develop a software system in a mid term. In the near future this new paradigm will obtain the respect that deserves.

We should remember that people thought the same with assembler languages and then with programming languages, so we have to wait to know if MDA will change the way we develop software. Anyways, it is a very interesting framework to study and we expect that our processor of OCL expressions could be a small contribution to MDA’s cause.

METAMODELING 3

3 METAMODELING

3.1 INTRODUCTION

Metamodeling is an activity that produces, among other things, metamodels. We can define a metamodel as a precise definition of the constructs and rules needed for creating models. It is possible to consider a metamodel as a model plan.

This chapter begins with an introduction to the different levels of representation of real-world concepts. Next step is to deal with metaschemas, the basic elements in metamodeling. As an example of metaschema we will show a simplified view of the UML metamodel.

Finally, we will introduce the concept of meta-metaschema, whose best-known representative is the Meta-Object Facility (MOF).

3.2 LEVELS OF REPRESENTATION

The classical framework for metamodeling is based on an architecture with four layers of representation (see [Obj02]). These layers are described as follows according to their abstraction level from bottom to top:

- The information layer is comprised of the data that we wish to describe or model.

- The model layer is comprised of the metadata that describes the data in the information layer. Such metadata is aggregated as models.

- The metamodel layer is comprised of the description that defines the structure and semantics of metadata. This meta-metadata is aggregated as metamodels.

- The meta-metamodel layer is comprised of the description of the structure and semantics of meta-metadata.

In the next figure we can see this four-layered architecture and the important concepts over it placed in their correct layer.

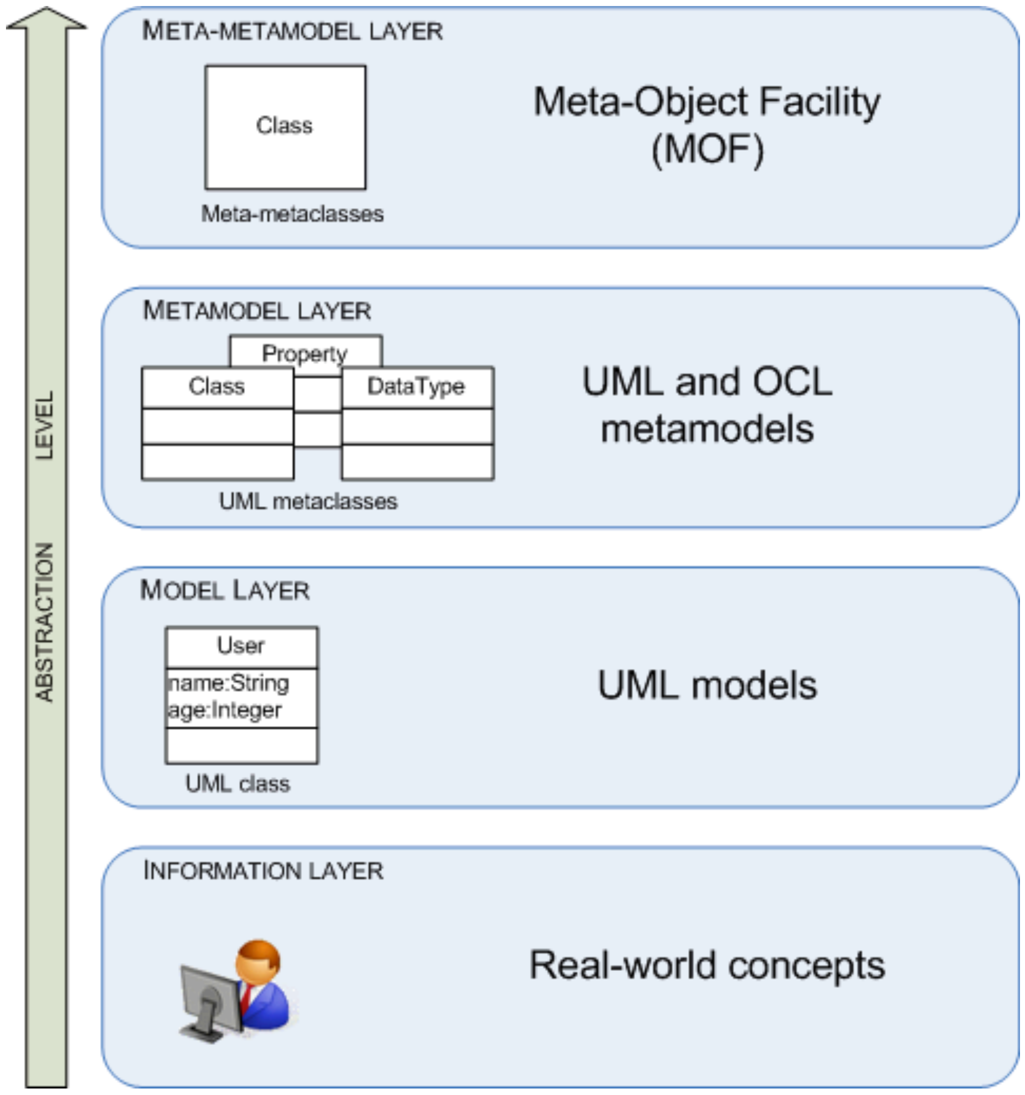


Figure 3.1: Representation levels

As we will see at next sections, UML classes are instances of the UML metamodel that is already an instance of the Meta-Object Facility meta-metamodel.

3.3 METASCHEMAS

At previous chapter we defined a conceptual schema as a representation of general knowledge about a domain. Therefore, a metaschema can be defined as a schema that represents knowledge about a schema.

Metaschema: a schema that represents knowledge about a schema.

Definition 3.1: Metaschema

The objects and relations of a domain are represented by symbols in a conceptual schema. Thus, such representation is an instance of another model in an upper level of abstraction. This model, or schema, contains information about the data in the first-level schema, so it is a metaschema.

The UML metamodel is the metaschema that we will explain in this chapter because it is the metaschema used in the Eina GMC tool and therefore in our OCL 2.0 expressions processor, due to UML is the modeling language used to write our models.

3.4 THE UML METAMODEL

Since our purpose is to parse OCL 2.0 expressions related with an UML schema and to instantiate them as instances of the OCL metamodel, to understand the UML metamodel is mandatory in order to know how an UML model is represented as instance of its metamodel.

In this section we will introduce a simplified version of the UML 2.0 metamodel in order to make easier the understandability of the elements within such metamodel. A full version can be found at the UML 2.0 superstructure [Obj07].

3.4.1 Diagrams

At this point we show the relevant diagrams of the UML 2.0 metamodel. It is important to note that blue-coloured classes are part of the OCL 2.0 metamodel that will be explained at next chapters.

3.4.1.1 Elements

Element is the base concept in the UML 2.0 metamodel. It generalizes all the components of the UML, as we will see in the next diagrams.

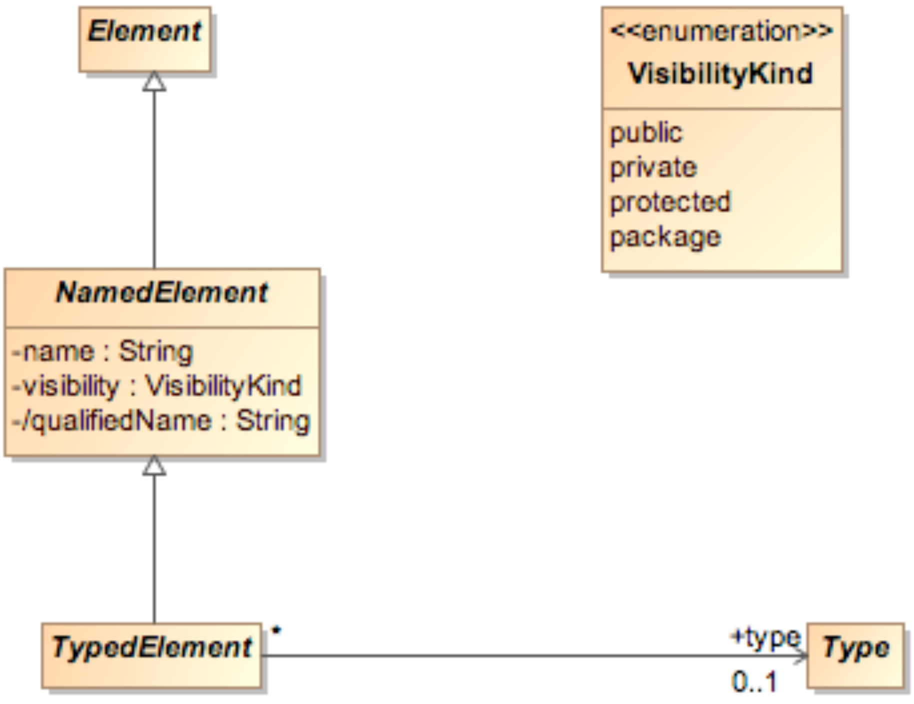


Figure 3.2: UML 2.0 metamodel. Elements

On the other hand, NamedElement class represents all the elements that can have a name. Furthermore, as a subclass of NamedElement we have TypedElement, which obviously represents all the elements that can have a name and a type. Type class is an abstract class that will be explained in a next diagram and represents the possible different types that a TypedElement can have.

3.4.1.2 Multiplicity elements

A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A multiplicity element embeds this information to specify the allowable cardinalities for an instantiation of this element.

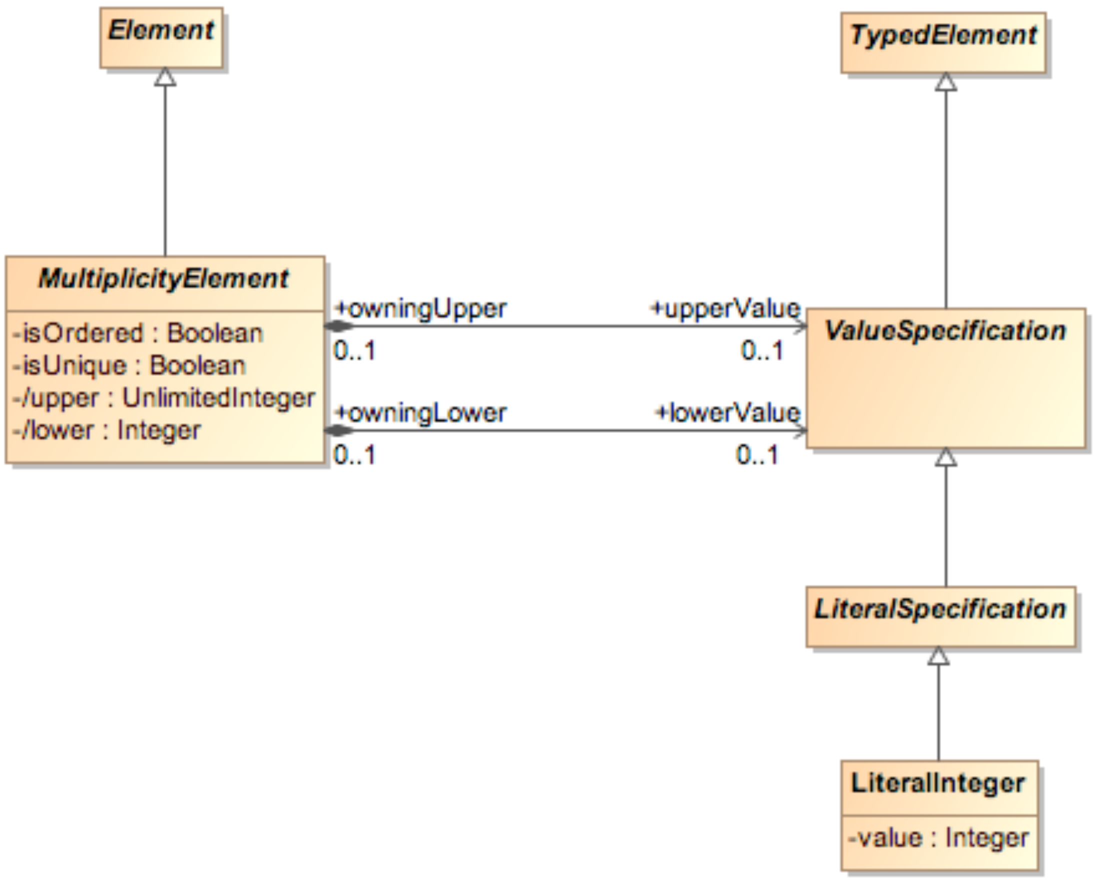


Figure 3.3: UML 2.0 metamodel. Multiplicity element

A MultiplicityElement has two relations with ValueSpecification element representing the upper and the lower multiplicities. In an association end 0..1 these two numbers are stored in these relations as instances of LiteralInteger metaclass. Note that such metamodel class has an Integer attribute where to place these multiplicity numbers.

In our special case of the Eina GMC tool, when an association end has a multiplicity of 1..*, the asterisk value is denoted with the Integer value -1.

3.4.1.3 Constraints

Next diagram deals with the representation of Constraints.

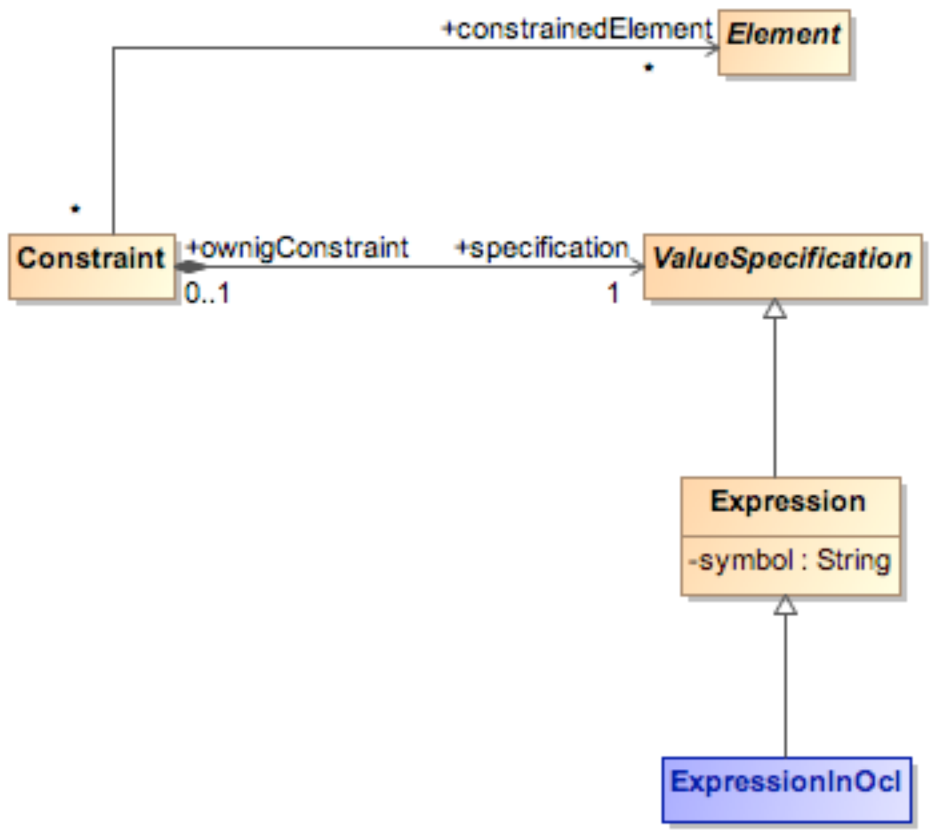


Figure 3.4: UML 2.0 metamodel. Constraint

This element is the nexus between UML and OCL metamodels. It contains a relation with an ExpressionInOcl metaclass that belongs to the OCL 2.0 metamodel. All constraints that can be defined are converted into instances of the OCL metamodel, and ExpressionInOcl is the wrapper class of them.

When OCL metamodel will be explained, such class will be described with its complete structure.

Furthermore, Constraint metaclass has a relation with general class Element, representing the element where the constraint must be applied.

3.4.1.4 Generalizations

A generalization is a hierarchical relation between two Classifier instances. One of these instances represents the general element while the other represents its subclass, or specific element.

For example, in a model hierarchy with two classes, Fruit and Banana, where Banana is subclass of Fruit, this relation is represented by means of a Generalization class whose general Classifier is Fruit and whose specific Classifier is Banana.

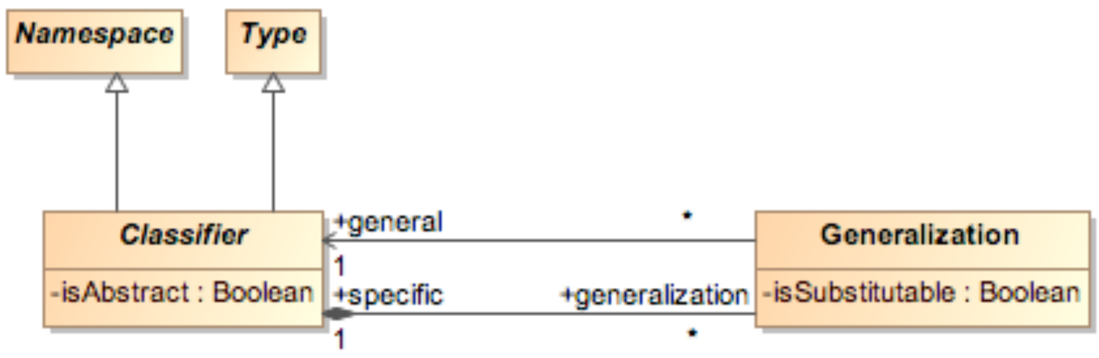


Figure 3.5: UML 2.0 metamodel. Generalization

Note that Classifier is a subclass of Type. So when Classifier subclasses will be explained we will emphasize that they can be used as types for other elements.

3.4.1.5 Operations

To represent operations the UML metamodel provides the Operation metaclass.

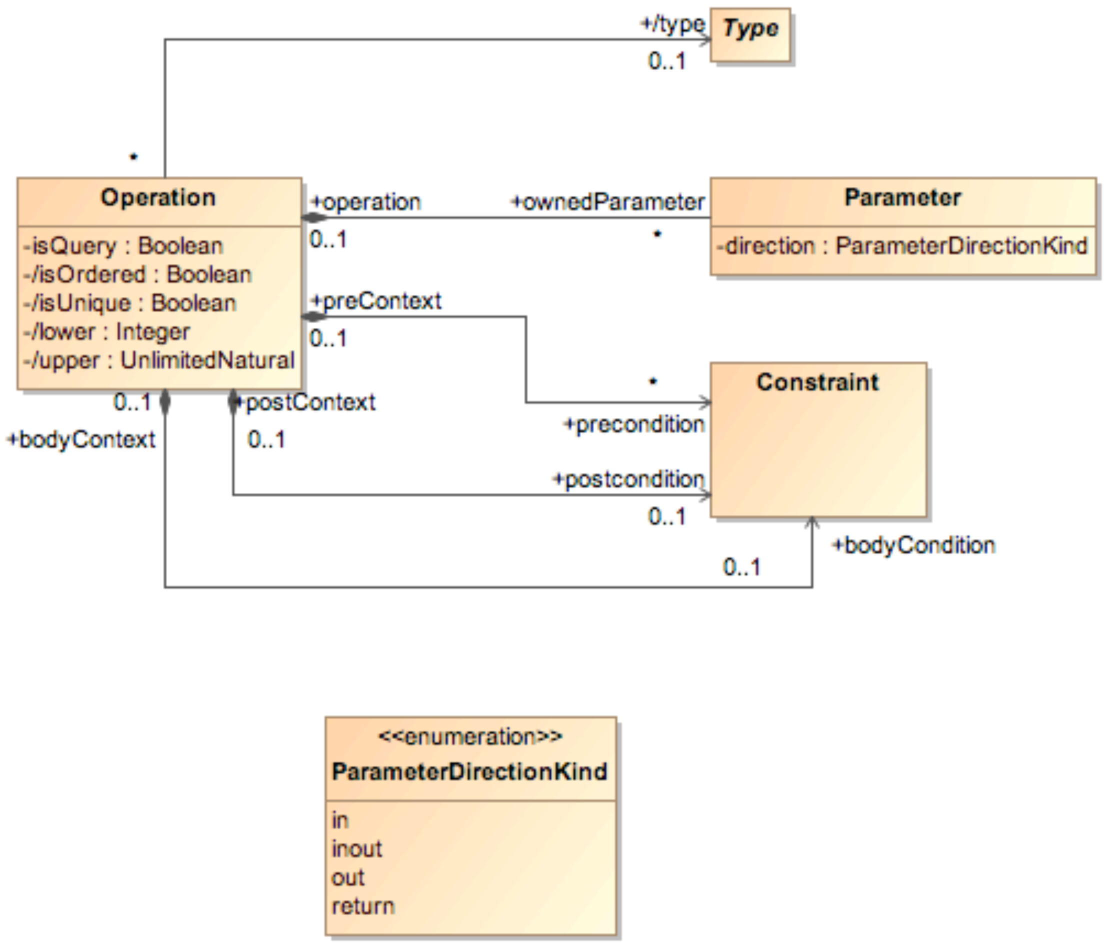


Figure 3.6: UML 2.0 metamodel. Operations

Such class has a multiple relation with the Parameter metaclass that represents all the parameters that an operation can own. Each Parameter instance has an attribute indicating the direction of the represented parameter, i.e., in, in-out, out or return.

Operation also has a derived (pay attention to the slash before the association end) relation with Type class. It is important to emphasize that it is derived due to the fact that the type of an Operation is equal to the type of its parameter that has a return direction, if it exists. So, to store the type of an operation we have to create an instance of the Parameter class with a return direction and such type.

Finally, Operation metaclass has another three relations with Constraint metaclass to store possible preconditions, postconditions or body expression written in OCL, as we will explain when the OCL metamodel will be explained.

3.4.1.6 Classes and associations

This is one of the most important diagrams in the UML metamodel.

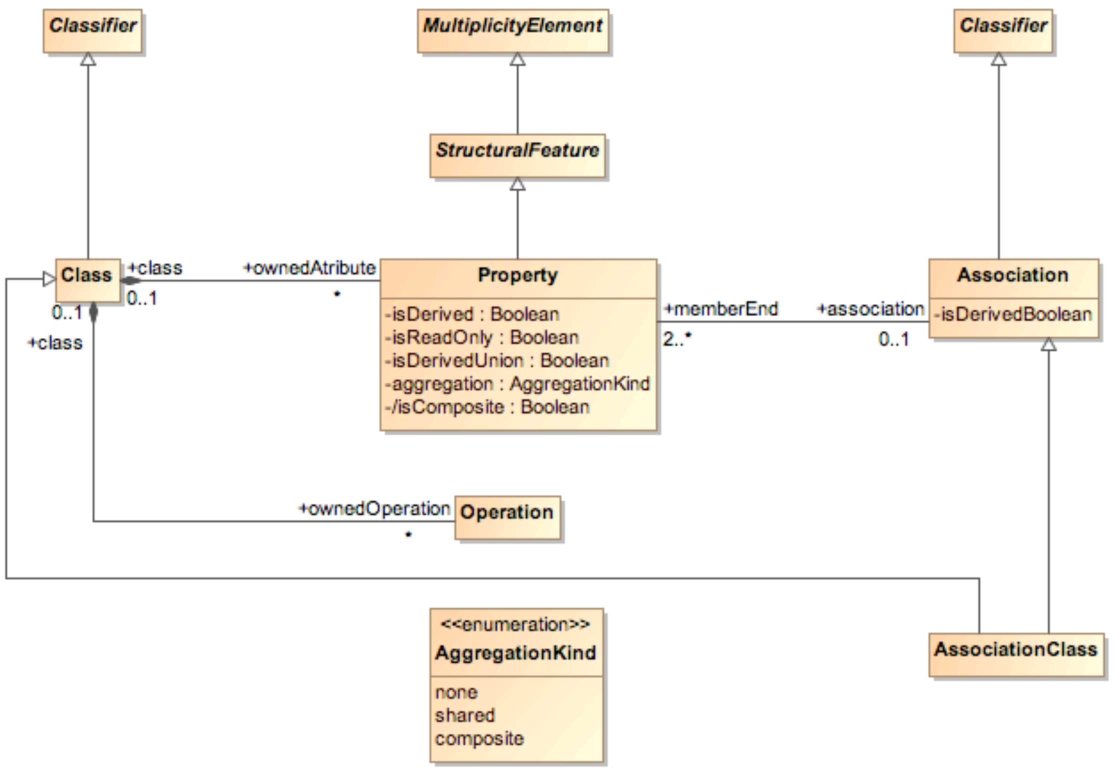


Figure 3.7: UML 2.0 Metamodel. Classes and associations

Class metaclass represents UML classes. Such metaclass has a relation with Property class that represents both the attributes owned by such class and the association ends that are part of an association with such class as a member. In concrete, it contains the association ends that are placed at such class end.

This is a change from UML version 1.1 because Property covers attributes and association ends. Before that an Attribute and an AssociationEnd metaclasses were within the metamodel.

Class also contains a relation with Operation class representing the operations owned by a UML class.

The last elements are Association and AssociationClass. First of them represents an association in a model and has relation with at least two Property instances representing the association ends that are members in such association.

Furthermore, we have AssociationClass that is subclass of both Association and Class. Therefore it represents an association (binary or with more than two members) that has an associative class. Since is a subclass of Class metaclass, AssociationClass can have the same relations with Property and Operation because they are directly inherited.

Finally, it is important to note that both Class and AssociationClass are also valid types to use because both are subclasses of Classifier, and therefore are indirectly subclasses of Type.

3.4.1.7 Data types

Our last diagram deals with data types and enumerations. DataType metaclass represents data types in the UML language. For example, Integer, Real, String or

Boolean types are instances of DataType int the UML language and are represented by such metaclass in the UML metamodel.

On the other hand, enumerations are special UML types that can own a set of elements specifying the values that such type can have. For example we can create an enumeration called Gender that can have two values Male and Female. Such enumeration is represented by the Enumeration metaclass, and its two values will be EnumerationLiteral instances linked with such Enumeration through the association shown in Figure 3.8.

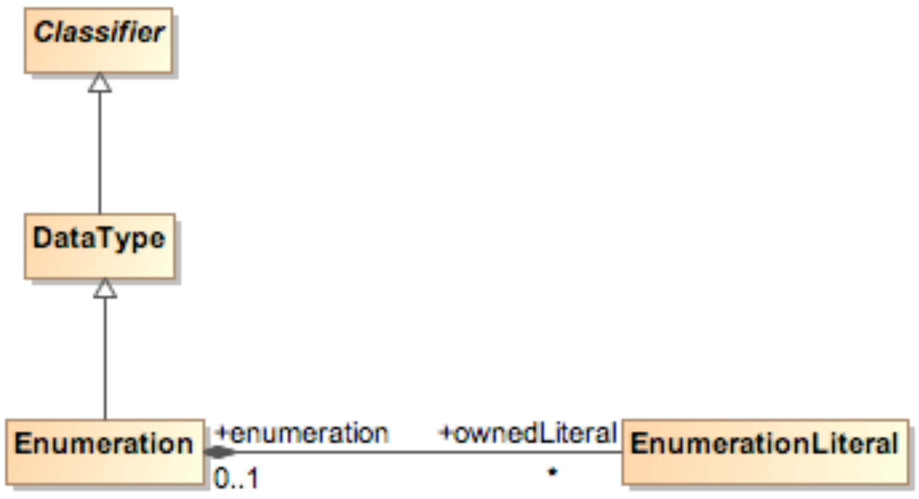


Figure 3.8: UML 2.0 metamodel. Data types

So, we have seen all the essential metaclasses of the UML 2.0 metamodel along this section. Some of them will be remembered when OCL 2.0 metamodel will be explained.

3.5 META-METASCHEMAS

A meta-metaschema is a schema that represents general knowledge about a domain consisting of a metaschema. The instances and relations in a metaschema are represented by symbols in the meta-metaschema.

Meta-metaschema: a schema representing general knowledge about a metaschema.

Definition 3.2: Meta-metaschema

So, meta-metaschemas are the highest level of representation of real-world concepts. Metaschemas and meta-metaschemas can be written in the same language or a different language.

The best-known meta-metaschema is the Meta-Object Facility (MOF) that will be introduced at next section.

3.6 THE META-OBJECT FACILITY (MOF)

The Meta-Object Facility is a meta-metaschema specified by the OMG [Obj02]. The UML metaschema is an instance of the MOF, but it can also be the meta-metaschema of other languages.

The MOF is also the basis of XMI, a metadata interchange language that will be described at next chapter.

Figure 3.9 shows a simplified version of the MOF model that is similar than the UML metamodel. MOF is smaller than the UML metaschema and both share concepts like for example Class metaclass. But in MOF, Class is a meta-metaclass whereas in the UML metaschema Class is an instance of the Class in MOF.

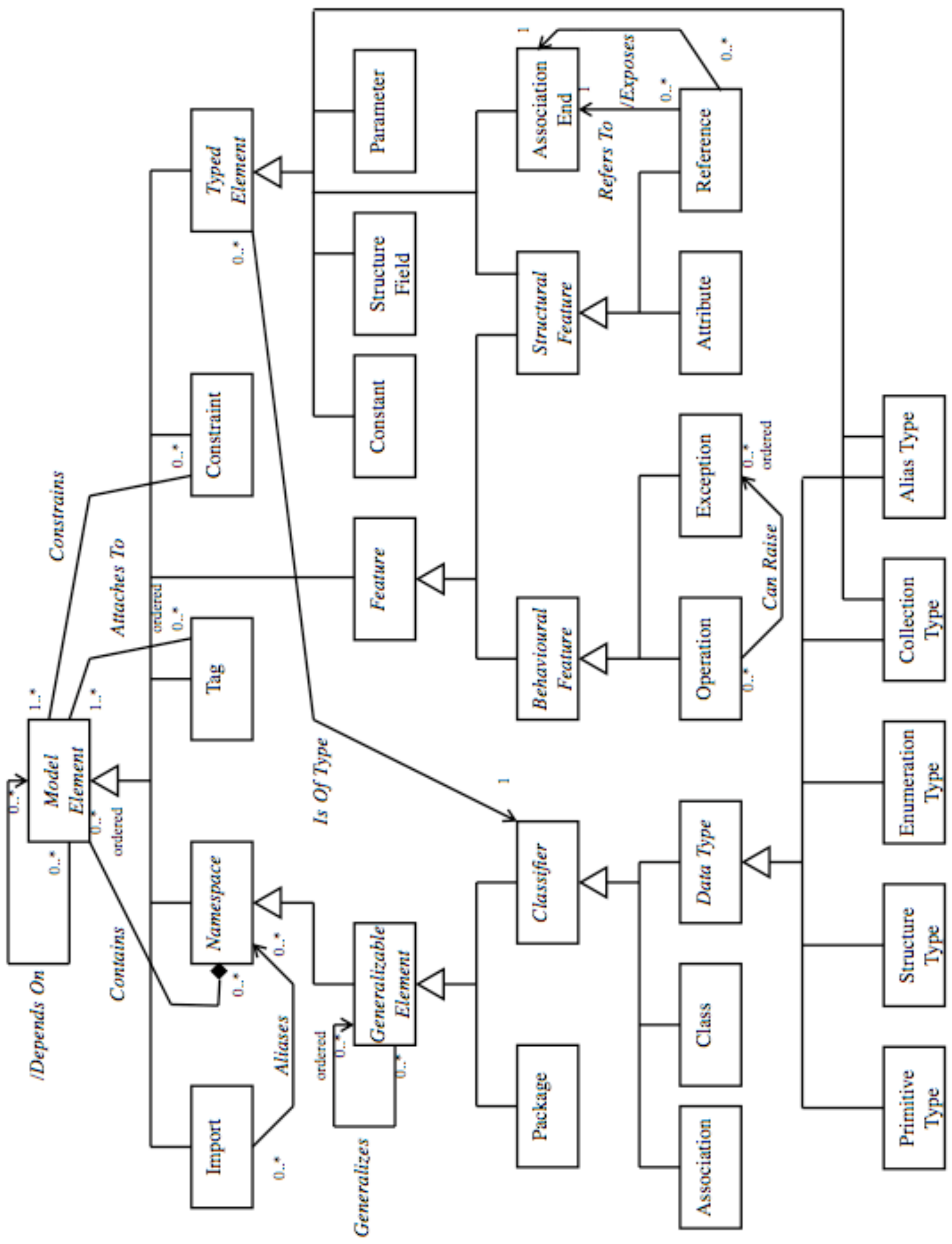


Figure 3.9: MOF metamodel extracted from [Obj02]

On the other hand, MOF cannot be used as conceptual modeling language because it lacks important features, such as association classes that are needed when modeling.

Therefore as a conclusion we have to remember that real-world concepts can be modelled using conceptual modeling languages like the UML. The resultant models of this modeling process are instances of a metaschema that specifies and describes the conceptual modeling language, like the UML metaschema for the UML language.

Finally, each metaschema is an instance of a meta-metaschema that is a general schema shared by most of languages. The use of meta-metaschemas allows conversion from a language to others. For example, if we have a model written in UML and we want to convert it into Entity-relationship (ER) model we only have to go up in the abstraction hierarchy until the MOF instantiation of such model and then convert such MOF representation into an ER instantiation going down. It is possible because both the UML metaschema and the ER metaschema may have the same meta-metaschema, i.e., the Meta-Object Facility.

XML METADATA INTERCHANGE 4

4 XML METADATA INTERCHANGE

4.1 THE NEED OF SHARING INFORMATION

We live in a changing society where the information is an essential keystone. Therefore sharing information is a task that we perform every day although we do not realize it.

In order to be successful when you are involved in a software project, the main task to do properly is the exchange of information between the different parts implicated on it.

In our scope, the conceptual schemas are the guidelines and construction plans of the building that represents our final software. So, to guarantee that this information is accessible and understandable for everyone is a priority job.

Understandability is provided by the Unified Modeling Language because it is a common standard language to model software systems. Moreover, to ensure the accessibility of the conceptual schemas we have the help of general representation languages that are very useful to share information between different formats of representation in order to simplify this process of interchange.

The most common representation language as standard of data storing and interchange language is the eXtensible Markup Language (XML), where is the basis for the XML Metadata Interchange (XMI). The object of this chapter is the study of the XMI.

4.2 XML AND ORIGINS

The Extensible Markup Language (XML, [XMLw]) is a general-purpose specification for creating custom markup languages.

A markup language is a set of annotations to text that describe how it is to be structured, laid out, or formatted. Markup languages have been in use for centuries, and in recent years have also been used in computer typesetting and word-processing systems.

XML is classified as an extensible language because it provides the functionality of defining own elements. Its primary purpose is to facilitate the sharing of structured data across different information systems, particularly via the Internet where has become in a standard.

XML is also used both to encode documents and to serialize data. Its facility to be processed makes both the data recovery and the data interchange process faster and simple for everyone.

XML was born as a simplified subset of the Standard Generalized Markup Language (SGML), which is a descendant of IBM's Generalized Markup Language (GML), created in the 1960s.

SGML was originally created to enable the sharing of machine-readable documents in very large projects. Nevertheless, XML was designed to be relatively human-readable.

As descendants of the XML, there has appeared a huge range of other languages based on it, like XHTML, RSS or XMI.

4.3 XML METADATA INTERCHANGE

The XML Metadata Interchange is a markup language descendant from XML. It is an OMG standard for exchanging metadata information.

When two or more systems need to share information, their designers have to agree with the format they use to represent such information in order to obtain a success communication.

XMI simplifies this job because it provides a standard format to represent and exchange data from a conceptual schema. For this reason, in our OCL 2.0 expressions processor we chose the XMI as a base format for to represent and serialize our models.

Furthermore, most Computer Aided Software Engineering (CASE) tools use this format to export the models made on them, and this situation implies that with some little changes such XMI files representing models could be used with our processor directly.

4.4 UML, MOF AND XMI

XML Metadata Interchange (XMI) is a standard for representing data about instances of types of MOF schemas in XML. Using XMI, two systems that share the same MOF schema can exchange data about its instances in a standard way, and no further explicit agreement is required.

In our case, we will use XMI to represent UML models through the UML metamodel, which is a descendant of MOF, as we seen at previous chapter.

XMI rules can be applied to obtain an XML representation of any set of instances of classes, attributes and associations of any schema whose classes, attributes, and associations are instances of the corresponding MOF classes.

Since the UML metaschema is a schema whose classes, attributes and associations are instances of the corresponding MOF classes, the instances of the UML metaschema can be represented using XMI. That is, we can use XMI rules to achieve a standard XML representation of UML schemas.

4.5 XMI REPRESENTATION OF UML SCHEMAS

A complete presentation of XMI can be found at [GDB02]. In this section we only show how to represent in XMI a conceptual schema made in UML.

The overall structure of an XMI document representing a UML model with classes and relations is:

```
<xmi:XMI xmi:version = “2.1” xmlns:xmi = “http://www.omg.org/XMI”>
   <!-- Classes and relations -->
</xmi:XMI>
```

Example 4.1: Overall structure of a XMI file

Note that all concepts inside a markup language like XMI are represented between angle brackets (‘<’ and ‘>’). Furthermore, such concepts are placed inside tags. A tag is a wrapper that starts with an open structure like <xmi:XMI … > and ends with a close one like </xmi:XMI>. It is important to emphasyze that all tags must have both open and close marks. Special tags starting with <!-- and ending with --> are considered comments.

All the concepts inside an outer concept are members of such concept. For instance, in the Example 4.1 all the concepts placed inside the xmi:XMI tag are members of such tag.

Within this hierarchical tree structure all elements must be placed according to their conceptual schema. Therefore, this structure is easy to process with a single-sweep depth algorithm.

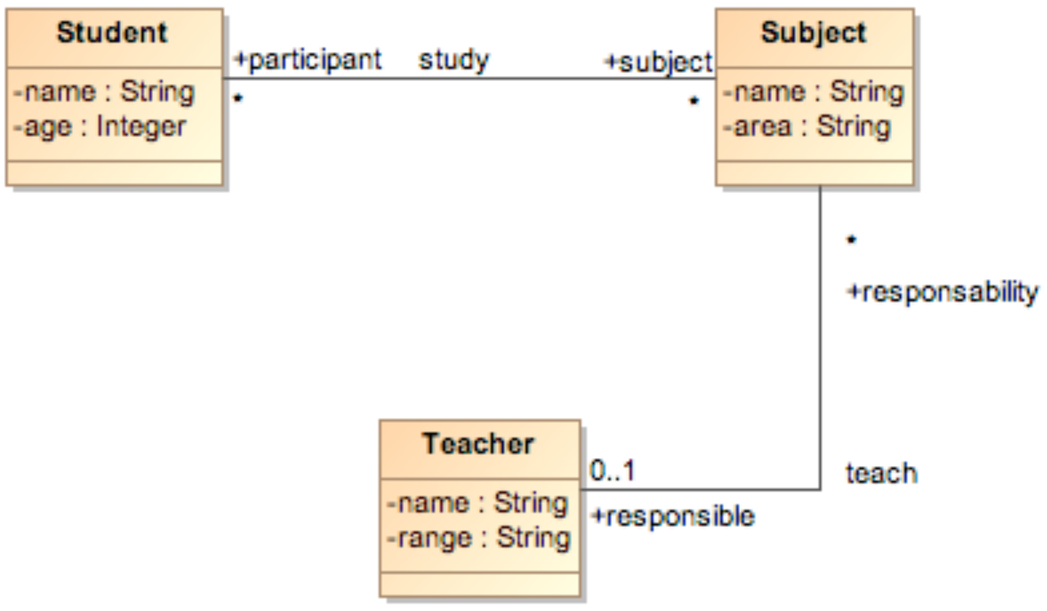


Figure 4.1: Simple conceptual schema

This previous conceptual model will be our example to know how to represent a conceptual schema in UML into an XML file according to the rules of the XMI.

First of all we will show the representation of the Student class with its owning attributes in XMI format. Note that both attributes have the reference of their type. In this case there are two DataType classes representing the String type and the Integer Type, respectively DT1 and DT2.

Furthermore, some extra information for each attribute is shown, like boolean attributes indicating if it is composite or derived.

```
<Class xmi:id = “C1” name = “Student” isAbstract = “false”>
 <ownedAttribute xmi:id = “A1” name = “name” lower = 1 upper = 1
    isComposite = “false” isDerived = “false”>
     <type xmi:type = ”DT1”/>
 </ownedAttribute>
 <ownedAttribute xmi:id = “A2” name = “age” lower = 1 upper = 1
    isComposite = “false” isDerived = “false”>
     <type xmi:type = ”DT2”/>
 </ownedAttribute>
</Class>
```

Example 4.2: XMI representation of an UML class

Once we have introduced a little part of our resulting XMI file, the next step is to show the whole file in order to study its particularities.

```
<xmi:XMI xmi:version = “2.1” xmlns:xmi = “http://www.omg.org/XMI”>
    <!—Data types -->
    <DataType xmi:id = “DT1” name = “String”>
    <DataType xmi:id = “DT2” name = “Integer”>
    <!-- Classes -->
    <Class xmi:id = “C1” name = “Student” isAbstract = “false”>
          <ownedAttribute xmi:id = “A1” name = “name” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT1”/>
          </ownedAttribute>
          <ownedAttribute xmi:id = “A2” name = “age” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT2”/>
          </ownedAttribute>
    </Class>
    <Class xmi:id = “C2” name = “Subject” isAbstract = “false”>
          <ownedAttribute xmi:id = “A3” name = “name” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT1”/>
          </ownedAttribute>
          <ownedAttribute xmi:id = “A4” name = “area” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT1”/>
          </ownedAttribute>
    </Class>
    <Class xmi:id = “C3” name = “Teacher” isAbstract = “false”>
          <ownedAttribute xmi:id = “A5” name = “name” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT1”/>
          </ownedAttribute>
          <ownedAttribute xmi:id = “A6” name = “range” lower = 1
          upper = 1 isComposite = “false” isDerived = “false”>
                <type xmi:type = ”DT1”/>
          </ownedAttribute>
    </Class>
    <!-- Relations -->
    <Association xmi:id = “AS1” name = “study” memberEnd = “PR1
    PR2”/>
    <Association xmi:id = “AS1” name = “teach” memberEnd = “PR3
    PR4”/>
    <Property xmi:id = “PR1” name = “participant” lower = 0 upper
    = * isComposite = “false” isDerived = “false” type = “C1”
    association = “AS1”/>
    <Property xmi:id = “PR2” name = “subject” lower = 0 upper = *
    isComposite = “false” isDerived = “false” type = “C2”
    association = “AS1”/>
    <Property xmi:id = “PR3” name = “responsible” lower = 0 upper
    = 1 isComposite = “false” isDerived = “false” type = “C3”
    association = “AS2”/>
    <Property xmi:id = “PR4” name = “responsibility” lower = 0
    upper = * isComposite = “false” isDerived = “false” type =
    “C2” association = “AS2”/>
</xmi:XMI>
```

Example 4.3: XMI representation of UML model shown at Figure 4.1

It is important to note that for each Property representing the association ends it is shown its multiplicity and the type of the UML class where it belongs. Furthermore, the identifier of the association where it participates is also shown.

In next chapters we will see more examples of conceptual schemes represented in XMI language, but we will see that they will have some differences in their syntax compared with the XMI shown here.

This is a common problem about standards. Although XMI is a standard, different people can interpret it in different ways, and therefore, this is the cause of the existence of some small differences in the syntax of the language. Furthermore, to use different version of the UML metamodel is another cause for differences in the XMI format.

As we will see, Eina GMC designers and programmers met this problem and to solve it, they implemented the XMI Converter. This software tool converts XMI files from CASE tools to the inner format of Eina GMC and vice versa.

To summarize, we have seen the possibilities of XMI and how to represent UML models into an XML file in order to serialize and share our model data. This representation is done according to a UML metaschema instance of MOF.

EINA GMC: A CONCEPTUAL MODELING ENVIRONMENT 5

5 EINA GMC: A CONCEPTUAL MODELING ENVIRONMENT

5.1 WHAT IS EINA GMC?

Eina GMC [EINw] is a project developed by members of the Research Group in Conceptual Modeling of Information Systems (GMC, [GMCw]) from the Technical University of Catalonia (UPC, [UPCw]) and the Open University of Catalonia (UOC, [UOCw]).

The aim of this project is to provide a full environment for working with conceptual schemas specified with both the Unified Modeling Language (UML) and the Object Constraint Language (OCL).

5.2 EINA GMC CORE

The best way of get in touch with the Eina GMC conceptual environment is through its Start Guide document that can be found at [GMC07]. All the concepts explained along this chapter are described in detail at such document and we consider highly recommendable to read it.

The Core of Eina GMC is a library written in Java, which helps working with conceptual schemas. It provides a set of classes implementing the UML 2.0 and OCL 2.0 metamodels.

A conceptual schema is an instantiation of UML and OCL metamodels. Therefore, using the Eina GMC Core it is possible to instantiate conceptual schemas as a set of

Java objects. Furthermore, these schemas can be stored in XMI files, as we will see when we explain the XMI Converter.

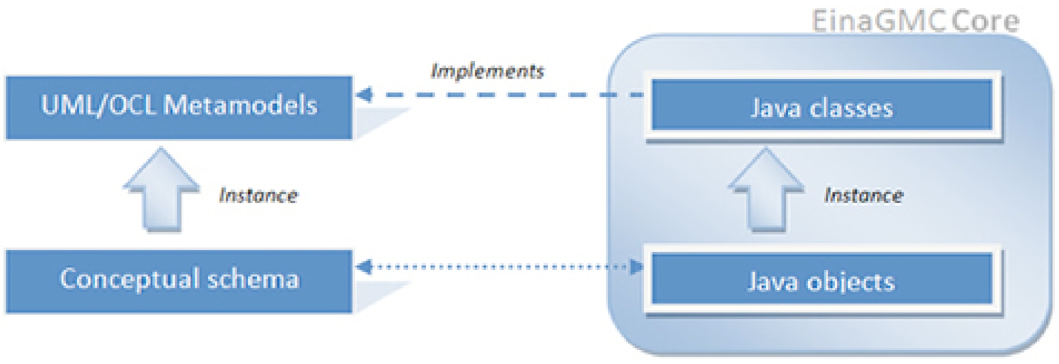


Figure 5.1: Eina GMC Core (Image from [EINw])

With the Eina GMC Core, the development of applications and functionalities related with conceptual modeling in a Java framework is a reality.

In our context, we are interested in processing OCL 2.0 expressions for to instantiate them into the OCL 2.0 metamodel, so this library allows us this behaviour.

As we explained earlier, our project is placed in the context of the Eina GMC project, and our aim is to extend this tool with the processing of OCL 2.0 expressions. Before the development of our project, if someone needs to instantiate an OCL 2.0 constraint into the Eina GMC conceptual environment metamodel, he or she has to program such conversion by instantiating each metaclass for each current constraint element.

Before our project, this person only has to write the OCL 2.0 constraint and call our processor with it. All the instantiation into the metamodel is made automatically and we can be centred only in conceptual modeling tasks, not in conversions or compilers programming.

Therefore, saving time is one of the consequences of our processor of OCL 2.0 expressions.

5.3 OTHER COMPONENTS

In this section we will explain the different components that conform the Eina GMC conceptual environment and complements the Core of such tool.

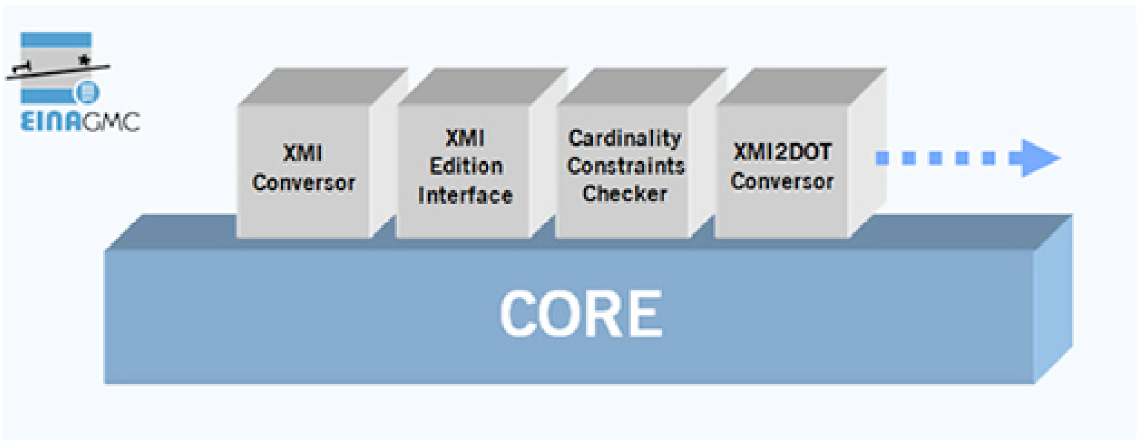


Figure 5.2: Components of the Eina GMC (Image from [EINw])

Our intention is not to show the details of these components because it can be found in [EINw]. We only want to explain what is the purpose of each one and also to give a brief description.

5.3.1 XMI CONVERTER

This is the most useful component of the Eina GMC, after the Core. It allows conversion between XMI files representing a conceptual schema from CASE tools format to Eina GMC format and vice versa. With this conversion we can see in a graphical way inside Poseidon conceptual schemas made by Eina GMC Core. In the current version the only CASE tool supported by now is Poseidon For UML.

5.3.2 XMI Edition Interface

The XMI Edition Interface is a graphical user interface (GUI) that allows editing XMI files representing conceptual schemas from Eina GMC in order to create new UML elements like classes, attributes or associations.

It allows us to avoid the programming task of doing changes in a conceptual schema.

5.3.3 Cardinality Constraints Checker

This component allows us to check the satisfiability of conceptual schemas expressed in the Eina GMC format. For example, to verify that there exists a set of instances that can be instantiated in a conceptual schema or if the cardinalities of a set of relations are correct.

5.3.4 XMI2DOT Converter

The XMI2DOT converter allows creating a DOT file with the graphical representation of the conceptual schema specified in an EinaGMC XMI file. Images (JPG, GIF, PNG...) can be generated from the DOT file using Graphviz tools. The current version allows converting a subset of UML elements, which is specified in the documentation of such component that can be found at [EINw].

5.4 HOW TO USE IT

Eina GMC can be used in two different ways. The simple way is using Eina GMC for drawing the conceptual model directly within Poseidon For UML [PFUw], and then export it to an XMI file and convert it to Eina GMC format through the XMI Converter component.

Once we have the saved XMI file, we can open it with Eina GMC in order to add changes or modifications with the XMI Edition Interface.

The complex way of using Eina GMC is by constructing the conceptual model and its instantiation directly with such library in a Java program. The Eina GMC conceptual environment provides a facade class for each UML and OCL metaclass that allows the necessary methods to create and instantiate such elements.

With such facades we can create a conceptual schema and the store it in an XMI file that can be shared or converted into the Poseidon format.

At next section we will study a simple conceptual schema specified in the UML language and the two existing methods to instantiate it into the UML metamodel of the Eina GMC tool.

5.5 AN EXAMPLE

To show how to use the Eina GMC tool we will start from a simple model and we will follow the necessary steps until obtain an XMI file representing such conceptual schema as instances of the UML metamodel.

First of all we present the model that represents our case study. It contains only five UML classes and one associative class, but we can see that a lot of common constructions appear on it.

We have a hierarchy of classes with Person class as general class and two subclasses under it, Employee and Client.

Furthermore, inside this conceptual schema there are two different relations, one of these has an associative class.

We chose this conceptual schema its constructions usually appear in most conceptual models that one can imagine.

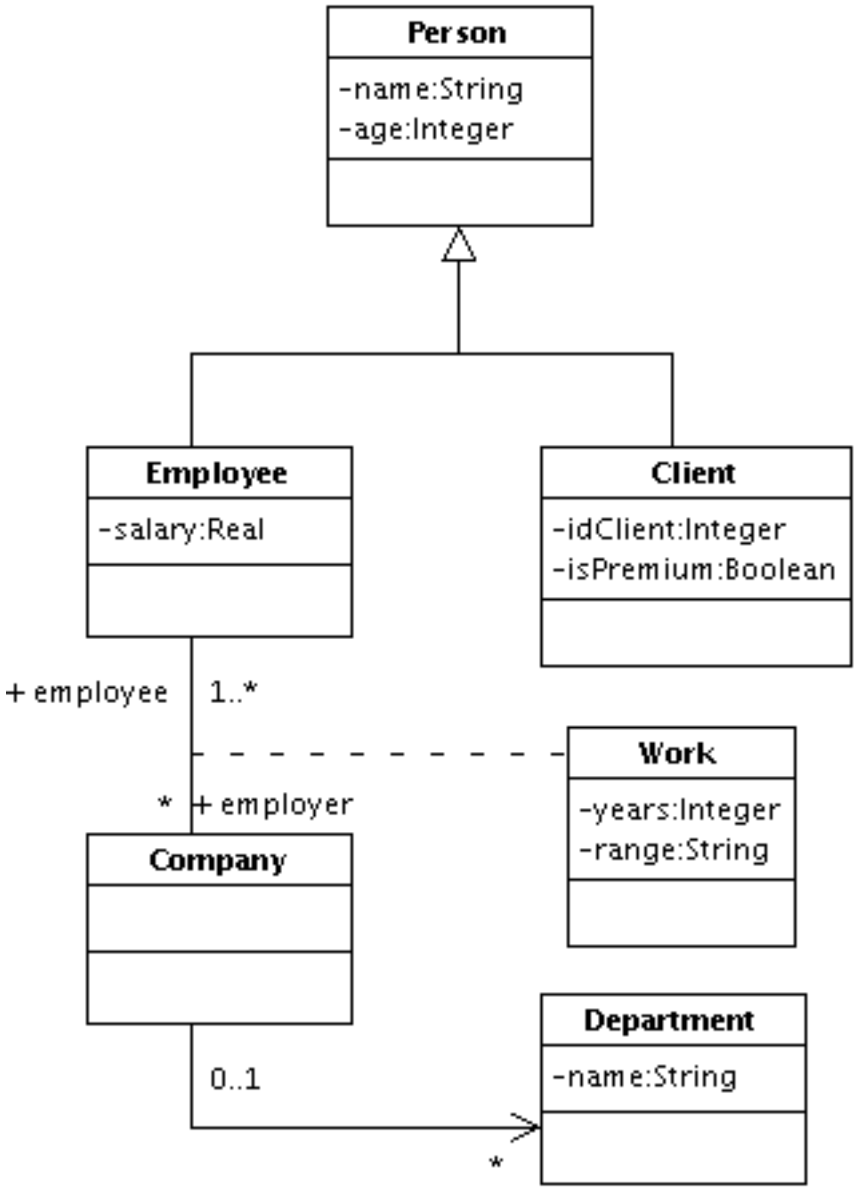


Figure 5.3: Conceptual model to practice with Eina GMC tool

5.5.1 Instantiation through Poseidon For UML

First of all it is important to own a copy of Poseidon For UML [PFUw] CASE tool installed in our computer.

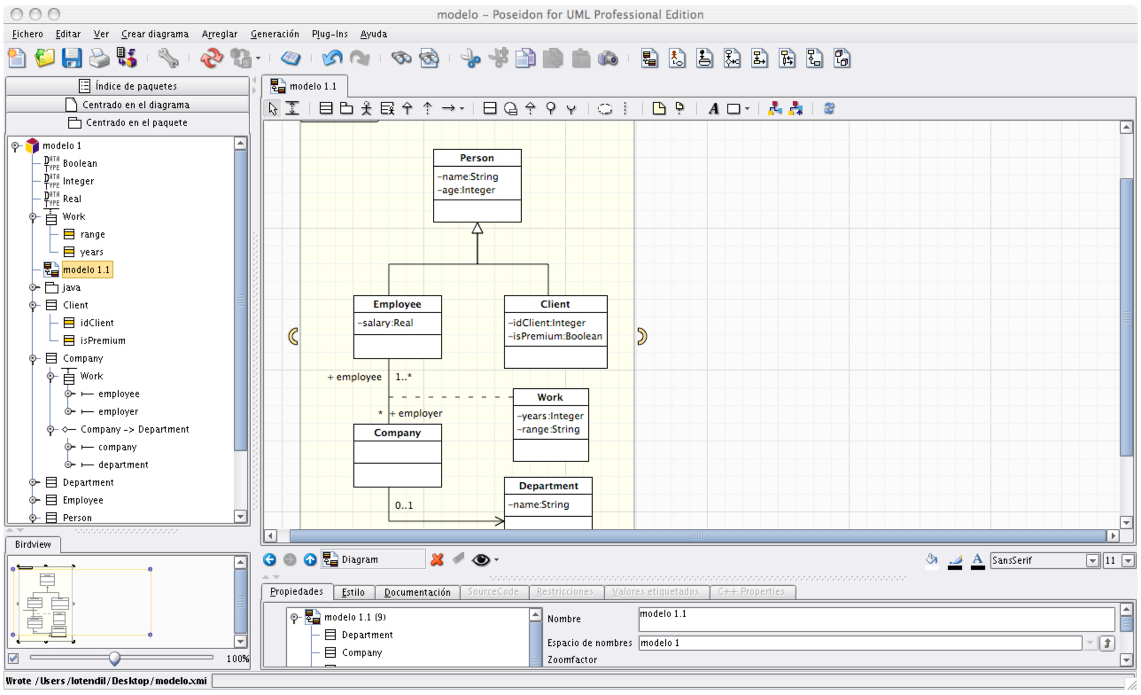


Figure 5.4: Poseidon For UML

With such tool we can create a class diagram representing the model shown at Figure 5.3.

Once we have modelled this conceptual schema, we have to export it to an XMI file. To do this task we have to open the File menu and select the "Export Project to XMI... " option.

Then, another window is open where we have to deselect the option that indicates if we want to store diagram information inside the XMI file.

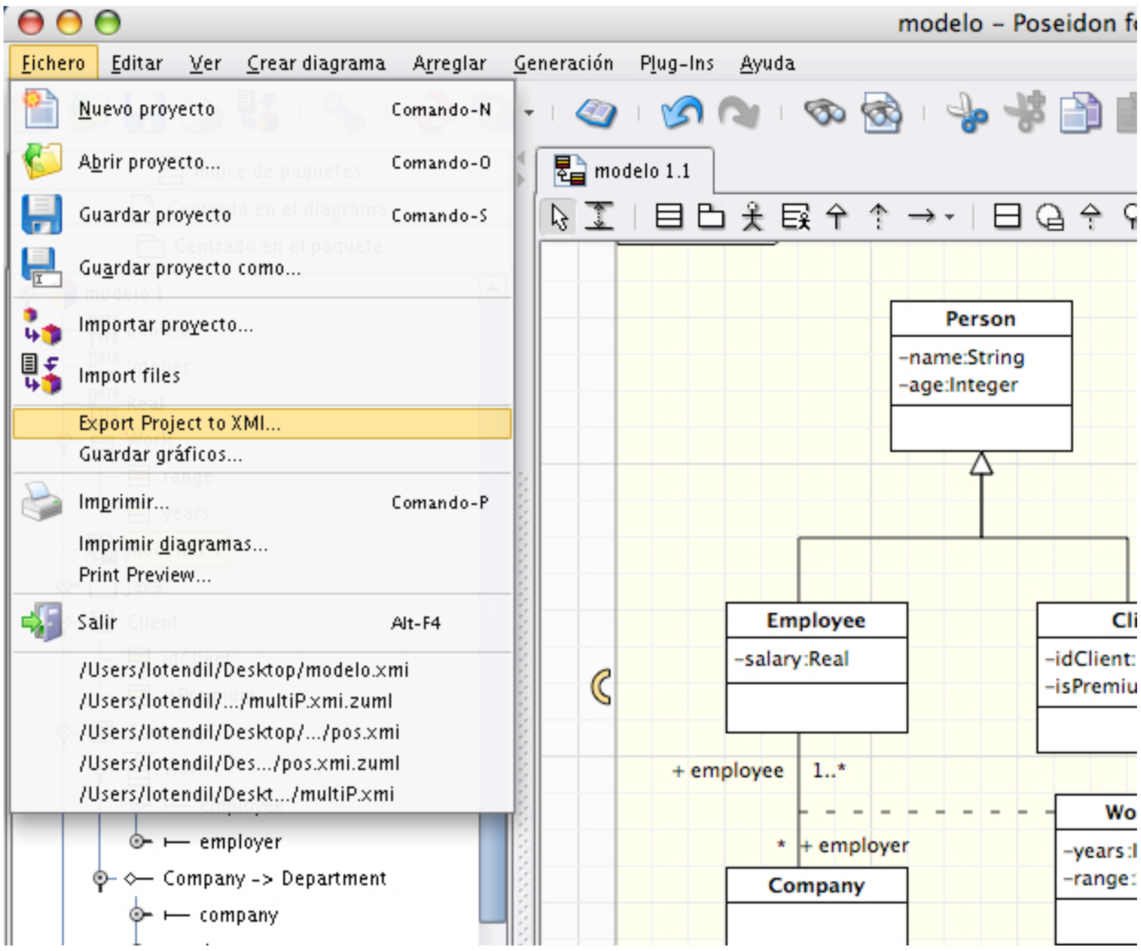
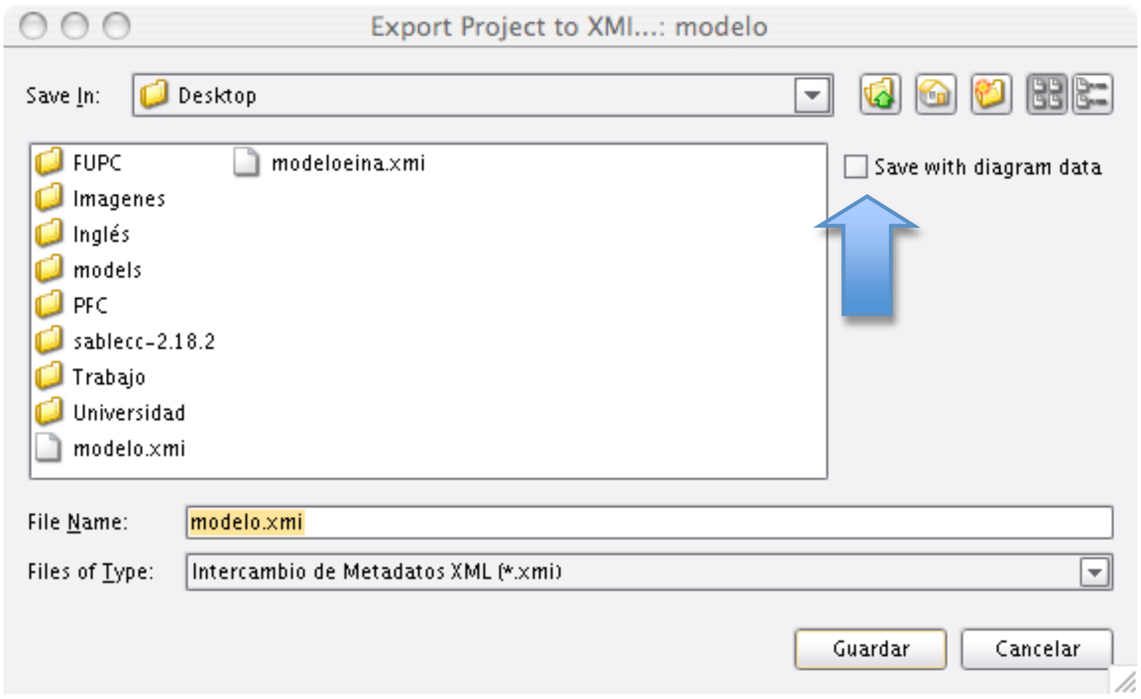


Figure 5.5: Export option inside Poseidon For UML

This diagram information consists in graphical information like how are placed the elements in the class diagram, so it does not contain relevant information, therefore we choose to avoid it.

Once we have exported the model into an XMI file the next step is to convert it into the Eina GMC format. To do this task we have the XMI Converter tool. To use it we only have to write a Java program that creates an instance of the ConverterToEinaGMC class and call the convert method as shown at Example 5.1.


Export Project to XMI...: modelo

Save In: Desktop

FUPC
Imagenes
Inglés
models
PFC
sablecc-2.18.2
Trabajo
Universidad
modelo.xmi
modeloeina.xmi

☐ Save with diagram data

File Name: modelo.xmi
Files of Type: Intercambio de Metadatos XML (*.xmi)

Figure 5.6: Export window inside Poseidon For UML

```
ConverterToEinaGMC ctgmc = new ConverterToEinaGMC();
ctgmc.convert("poseidon.xmi", "eina.xmi", "Poseidon");
```

where poseidon.xmi is the XMI file created on Poseidon For UML and eina.xmi is the output file in Eina GMC format.

Example 5.1: Conversion from Poseidon For UML to Eina GMC format

The last step is to open the new XMI file with the Eina GMC conceptual environment. We only have to create an instance of Project class and import the XMI file with the importXMI(“xmi-path”) method applied to such instance.

It is important to note that to do these steps we need a copy of the Eina GMC library and the XMIConverter in our computer. These libraries can be found at [EINw].

5.5.2 DIRECT INSTANTIATION

As we explained earlier, the complex way of instantiating a conceptual schema is creating each UML element inside such schema through the facade classes of the Eina GMC library.

At next example we will see how to use such facades in order to instantiate the model shown at Figure 5.3.

It is important to note that to specify a multiplicity through direct instantiation we have to know that infinite cardinalities are specified with a -1 number representing the * symbol in the XMI format for the Eina GMC tool. Therefore, an association end with cardinality 1..* is represented by values 1, -1.

```java
public static void makeModel() throws Exception {
    Project p = new Project();

    // Facade classes (Eina GMC uses UmlClass instead of Class
    // because Class is a reserved word in Java)
    UmlClassFacade ucf         = new UmlClassFacade(p);
    PropertyFacade pf          = new PropertyFacade(p);
    DataTypeFacade dtf         = new DataTypeFacade(p);
    LiteralIntegerFacade lif   = new LiteralIntegerFacade(p);

    // Creation of the basic types
    DataType integer           = dtf.createDataType();
    DataType real              = dtf.createDataType();
    DataType booleantype       = dtf.createDataType();
    DataType string            = dtf.createDataType();
    integer.setName("Integer");
    real.setName("Real");
    booleantype.setName("Boolean");
    string.setName("String");

    // Creation of the UML Classes
    UmlClass person            = ucf.createUmlClass();
    UmlClass employee          = ucf.createUmlClass();
    UmlClass client            = ucf.createUmlClass();
    UmlClass company           = ucf.createUmlClass();
    UmlClass department        = ucf.createUmlClass();
    person.setName("Person");
    employee.setName("Employee");
    client.setName("Client");
```

```
company.setName("Company");
department.setName("Department");

// Creation of the owned attributes for the previous classes
Property personName       = pf.createProperty();
Property personAge        = pf.createProperty();
Property employeeSalary   = pf.createProperty();
Property clientId         = pf.createProperty();
Property clientIsPremium  = pf.createProperty();
Property departmentName   = pf.createProperty();

// Set name and type
personName.setName("name");
personName.setType(string);
personAge.setName("age");
personAge.setType(integer);
employeeSalary.setName("salary");
employeeSalary.setType(real);
clientId.setName("idClient");
clientId.setType(integer);
clientIsPremium.setName("isPremium");
clientIsPremium.setType(booleantype);
departmentName.setName("name");
departmentName.setType(string);

// Place attributes into their owner UML class
personName.setUmlclass(person);
personAge.setUmlclass(person);
employeeSalary.setUmlclass(employee);
clientId.setUmlclass(client);
clientIsPremium.setUmlclass(client);
departmentName.setUmlclass(department);

// Creation of the Person hierarchy
GeneralizationFacade gf = new GeneralizationFacade(p);
gf.createGeneralization(person, client);
gf.createGeneralization(person, employee);

// Create association between Company and Department
AssociationFacade af = new AssociationFacade(p);
//     These are the two association ends
Property companyEnd       = pf.createProperty();
Property departmentEnd    = pf.createProperty();
companyEnd.setType(company);
departmentEnd.setType(department);

//     Create and set the multiplicity 0..1 to Company end
LiteralInteger lowermultiplicity = lif.createLiteralInteger();
LiteralInteger uppermultiplicity = lif.createLiteralInteger();
lowermultiplicity.setValue(0);
lowermultiplicity.setVisibility(VisibilityKindEnum.PUBLIC);
uppermultiplicity.setValue(1);
uppermultiplicity.setVisibility(VisibilityKindEnum.PUBLIC);
```

```
companyEnd.setLowerValue(lowermultiplicity);
companyEnd.setUpperValue(uppermultiplicity);

//     Create and set the multiplicity 0..* to Department end
LiteralInteger lowmultiplicity2 = lif.createLiteralInteger();
LiteralInteger uppmultiplicity2 = lif.createLiteralInteger();
lowmultiplicity2.setValue(0);
lowmultiplicity2.setVisibility(VisibilityKindEnum.PUBLIC);
uppermultiplicity2.setValue(-1); // -1 indicates *
uppermultiplicity2.setVisibility(VisibilityKindEnum.PUBLIC);
departmentEnd.setLowerValue(lowmultiplicity2);
departmentEnd.setUpperValue(uppmultiplicity2);

LinkedList ends = new LinkedList();
ends.add(companyEnd);
ends.add(departmentEnd);
Association companytodepartment = af.createAssociation(ends);

// Create association between employee and company with
// associative class Work
AssociationClassFacade acf = new AssociationClassFacade(p);

//     These are the two association ends
Property companyEnd2 = pf.createProperty();
Property employeeEnd = pf.createProperty();
companyEnd2.setType(company);
companyEnd2.setName("employer");
employeeEnd.setType(employee);
employeeEnd.setName("employee");

       // Create and set the multiplicity 0..* to Company end
LiteralInteger lowmultiplicity3 = lif.createLiteralInteger();
LiteralInteger uppmultiplicity3 = lif.createLiteralInteger();
lowmultiplicity3.setValue(0);
lowmultiplicity3.setVisibility(VisibilityKindEnum.PUBLIC);
uppmultiplicity3.setValue(-1); // -1 indicates *
uppmultiplicity3.setVisibility(VisibilityKindEnum.PUBLIC);
companyEnd2.setLowerValue(lowmultiplicity3);
companyEnd2.setUpperValue(uppmultiplicity3);

       // Create and set the multiplicity 1..* to Employee end
LiteralInteger lowmultiplicity4 = lif.createLiteralInteger();
LiteralInteger uppmultiplicity4 = lif.createLiteralInteger();
lowmultiplicity4.setValue(1);
lowmultiplicity4.setVisibility(VisibilityKindEnum.PUBLIC);
uppmultiplicity4.setValue(-1); // -1 indicates *
uppmultiplicity4.setVisibility(VisibilityKindEnum.PUBLIC);
employeeEnd.setLowerValue(lowmultiplicity4);
employeeEnd.setUpperValue(uppmultiplicity4);

LinkedList ends2 = new LinkedList();
ends2.add(companyEnd2);
ends2.add(employeeEnd);
```

```
    AssociationClass work = acf.createAssociationClass(ends2);
    work.setName("Work");

    // Create the owned elements of Work associative class
    Property years = pf.createProperty();
    Property range = pf.createProperty();
    years.setName("years");
    years.setType(integer);
    years.setUmlclass(work);
    range.setName("range");
    range.setType(string);
    range.setUmlclass(work);

    // Store the conceptual schema into an XMI file
    p.saveXMI("/Documents/models/newmodel.xmi");
    p.closeProject();
}
```

Example 5.2: Java program that instantiates the conceptual schema found at Figure 5.3

It is important to emphasize that in the same way we have created our model we are able to modify it with the setter methods that exists on each element class.

5.5.3 XMI OUTPUT FILE

Finally, the same output XMI file for both methods should have the similar content than as follows.

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version = '1.2' xmlns:uml = 'org.omg.xmi.namespace.uml'
timestamp = 'Fri May 02 18:27:15 CEST 2008'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
      <XMI.exporterVersion>1.0</XMI.exporterVersion>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <uml2.Kernel.Property xmi.id = 'a1' name = 'employee'>
      <uml2.Kernel.MultiplicityElement.lowerValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a2' visibility = 'public'
          value = '1'/>
      </uml2.Kernel.MultiplicityElement.lowerValue>
      <uml2.Kernel.MultiplicityElement.upperValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a3' visibility = 'public'
          value = '-1'/>
      </uml2.Kernel.MultiplicityElement.upperValue>
      <uml2.Kernel.TypedElement.type>
```

```
        <uml2.Kernel.Class xmi.idref = 'a4'/>
      </uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Property.association>
        <uml2.AssociationClasses.AssociationClass xmi.idref = 'a5'/>
      </uml2.Kernel.Property.association>
    </uml2.Kernel.Property>
    <uml2.Kernel.Property xmi.id = 'a6' name = 'employer'>
      <uml2.Kernel.MultiplicityElement.lowerValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a7' visibility = 'public'
           value = '0'/>
      </uml2.Kernel.MultiplicityElement.lowerValue>
      <uml2.Kernel.MultiplicityElement.upperValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a8' visibility = 'public'
           value = '-1'/>
      </uml2.Kernel.MultiplicityElement.upperValue>
      <uml2.Kernel.TypedElement.type>
        <uml2.Kernel.Class xmi.idref = 'a9'/>
      </uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Property.association>
        <uml2.AssociationClasses.AssociationClass xmi.idref = 'a5'/>
      </uml2.Kernel.Property.association>
    </uml2.Kernel.Property>
    <uml2.Kernel.Property xmi.id = 'a10'>
      <uml2.Kernel.MultiplicityElement.lowerValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a11' visibility = 'public'
           value = '0'/>
      </uml2.Kernel.MultiplicityElement.lowerValue>
      <uml2.Kernel.MultiplicityElement.upperValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a12' visibility = 'public'
           value = '-1'/>
      </uml2.Kernel.MultiplicityElement.upperValue>
      <uml2.Kernel.TypedElement.type>
        <uml2.Kernel.Class xmi.idref = 'a13'/>
      </uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Property.association>
        <uml2.Kernel.Association xmi.idref = 'a14'/>
      </uml2.Kernel.Property.association>
    </uml2.Kernel.Property>
    <uml2.Kernel.Property xmi.id = 'a15'>
      <uml2.Kernel.MultiplicityElement.lowerValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a16' visibility = 'public'
           value = '0'/>
      </uml2.Kernel.MultiplicityElement.lowerValue>
      <uml2.Kernel.MultiplicityElement.upperValue>
        <uml2.Kernel.LiteralInteger xmi.id = 'a17' visibility = 'public'
           value = '1'/>
      </uml2.Kernel.MultiplicityElement.upperValue>
      <uml2.Kernel.TypedElement.type>
        <uml2.Kernel.Class xmi.idref = 'a9'/>
      </uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Property.association>
        <uml2.Kernel.Association xmi.idref = 'a14'/>
      </uml2.Kernel.Property.association>
    </uml2.Kernel.Property>
    <uml2.Kernel.DataType xmi.id = 'a18' name = 'String'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a19'/>
        <uml2.Kernel.Property xmi.idref = 'a20'/>
        <uml2.Kernel.Property xmi.idref = 'a21'/>
      </uml2.Kernel.Type._typedElementOfType>
    </uml2.Kernel.DataType>
    <uml2.Kernel.DataType xmi.id = 'a22' name = 'Boolean'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a23'/>
      </uml2.Kernel.Type._typedElementOfType>
    </uml2.Kernel.DataType>
    <uml2.Kernel.DataType xmi.id = 'a24' name = 'Real'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a25'/>
      </uml2.Kernel.Type._typedElementOfType>
    </uml2.Kernel.DataType>
    <uml2.Kernel.DataType xmi.id = 'a26' name = 'Integer'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a27'/>
        <uml2.Kernel.Property xmi.idref = 'a28'/>
        <uml2.Kernel.Property xmi.idref = 'a29'/>
      </uml2.Kernel.Type._typedElementOfType>
    </uml2.Kernel.DataType>
    <uml2.Kernel.Class xmi.id = 'a13' name = 'Department'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a10'/>
      </uml2.Kernel.Type._typedElementOfType>
      <uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Property xmi.id = 'a20' name = 'name'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a18'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
      </uml2.Kernel.Class.ownedAttribute>
    </uml2.Kernel.Class>
    <uml2.Kernel.Class xmi.id = 'a9' name = 'Company'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a6'/>
        <uml2.Kernel.Property xmi.idref = 'a15'/>
      </uml2.Kernel.Type._typedElementOfType>
    </uml2.Kernel.Class>
    <uml2.Kernel.Class xmi.id = 'a30' name = 'Client'>
      <uml2.Kernel.Classifier.generalization>
        <uml2.Kernel.Generalization xmi.id = 'a31'>
          <uml2.Kernel.Generalization.general>
            <uml2.Kernel.Class xmi.idref = 'a32'/>
          </uml2.Kernel.Generalization.general>
        </uml2.Kernel.Generalization>
      </uml2.Kernel.Classifier.generalization>
      <uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Property xmi.id = 'a28' name = 'idClient'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a26'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
        <uml2.Kernel.Property xmi.id = 'a23' name = 'isPremium'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a22'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
      </uml2.Kernel.Class.ownedAttribute>
    </uml2.Kernel.Class>
    <uml2.Kernel.Class xmi.id = 'a4' name = 'Employee'>
      <uml2.Kernel.Type._typedElementOfType>
        <uml2.Kernel.Property xmi.idref = 'a1'/>
      </uml2.Kernel.Type._typedElementOfType>
      <uml2.Kernel.Classifier.generalization>
        <uml2.Kernel.Generalization xmi.id = 'a33'>
          <uml2.Kernel.Generalization.general>
            <uml2.Kernel.Class xmi.idref = 'a32'/>
          </uml2.Kernel.Generalization.general>
        </uml2.Kernel.Generalization>
      </uml2.Kernel.Classifier.generalization>
      <uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Property xmi.id = 'a25' name = 'salary'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a24'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
      </uml2.Kernel.Class.ownedAttribute>
    </uml2.Kernel.Class>
    <uml2.Kernel.Class xmi.id = 'a32' name = 'Person'>
      <uml2.Kernel.Classifier._generalizationOfGeneral>
        <uml2.Kernel.Generalization xmi.idref = 'a33'/>
        <uml2.Kernel.Generalization xmi.idref = 'a31'/>
      </uml2.Kernel.Classifier._generalizationOfGeneral>
      <uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Property xmi.id = 'a21' name = 'name'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a18'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
        <uml2.Kernel.Property xmi.id = 'a29' name = 'age'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a26'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
      </uml2.Kernel.Class.ownedAttribute>
    </uml2.Kernel.Class>
    <uml2.Kernel.Association xmi.id = 'a14'>
      <uml2.Kernel.Association.memberEnd>
        <uml2.Kernel.Property xmi.idref = 'a15'/>
        <uml2.Kernel.Property xmi.idref = 'a10'/>
      </uml2.Kernel.Association.memberEnd>
    </uml2.Kernel.Association>
    <uml2.AssociationClasses.AssociationClass xmi.id = 'a5' name='Work'>
      <uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Property xmi.id = 'a27' name = 'years'>
          <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'a26'/>
          </uml2.Kernel.TypedElement.type>
        </uml2.Kernel.Property>
        <uml2.Kernel.Property xmi.id = 'a19' name = 'range'>
          <uml2.Kernel.TypedElement.type>
```

```
              <uml2.Kernel.DataType xmi.idref = 'a18'/>
            </uml2.Kernel.TypedElement.type>
          </uml2.Kernel.Property>
        </uml2.Kernel.Class.ownedAttribute>
        <uml2.Kernel.Association.memberEnd>
          <uml2.Kernel.Property xmi.idref = 'a6'/>
          <uml2.Kernel.Property xmi.idref = 'a1'/>
        </uml2.Kernel.Association.memberEnd>
      </uml2.AssociationClasses.AssociationClass>
    </XMI.content>
  </XMI>
```

Example 5.3: XMI file relative to Figure 5.3 model

It is important to note that some fields (e.g., isAbstract, isDerived, isStatic, and others) that appear into the real XMI file have been deleted here in order to improve the understandability of the XMI.

THE OBJECT CONSTRAINT LANGUAGE 6

6 THE OBJECT CONSTRAINT LANGUAGE

6.1 WHAT IS THE OCL?

The Object Constraint Language (OCL, [Obj06]) is a modeling language defined within the Unified Modeling Language (UML, [Obj07]). It is the standard for specifying expressions that complete the information of object-oriented models and other software artifacts.

Initially, OCL was only a formal specification language extension to UML but now it can be used with other modeling languages beyond UML.

OCL is a descendant of Syntropy, an object-oriented modelling language, and it was firstly used in a business-modelling project within IBM. Its design is based on both formality and simplicity.

The OCL syntax is very understandable and it is easy to use for everyone familiar with programming language concepts, according to its design as a declarative language.

This language was firstly added to UML at version 1.1 and was restricted to constraints definition. In UML 2.0, it was extended in order to provide support for defining queries, pre and post conditions, derivation rules or initializations.

To conclude this introduction, we can affirm that without OCL, the application of the Model Driven Architecture is bound to fail, because using UML alone is not enough.

6.2 A CONCEPTUAL SCHEMA EXAMPLE

In this chapter we will introduce the whole allowed constructions defined at the OCL formal specification in its version 2.0. The example of Figure 6.1 will be used to illustrate the features of the language.

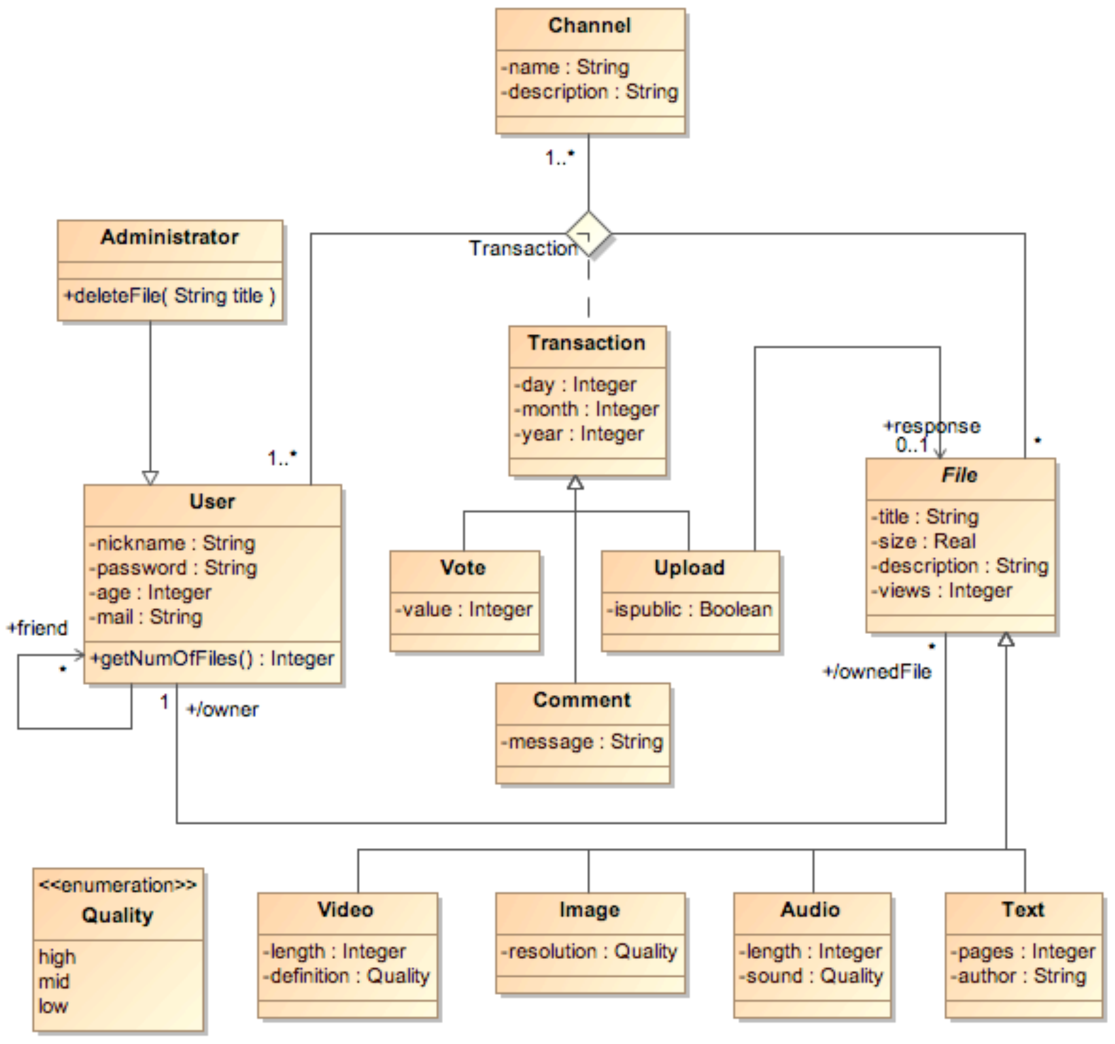


Figure 6.1: Model for the upload and share system example

Nowadays, due to the growing proliferation of web sites where to upload and share multimedia files like videos, photos or audio files, we decided to introduce a possible conceptual schema describing one of such web systems as our base example. Such schema can be shown in Figure 6.1.

At this diagram we can find a hierarchy of files representing the multimedia files that can be present into the system. We introduce four types of subclasses of File abstract class: Video, Audio, Image and Text. These are the different files to upload and share.

Furthermore, the User class represents the users of the system, and maintains a derived relation with File class in order to indicate which user is the owner of each file, i.e., the one that uploads such file.

Between File, User and Channel classes there is a ternary association with an associative class representing the transactions that users can do. There are three kinds of transactions and every one store the current data when is executed. Note that the representation of the ternary association is made by means of a rhombus shape.

Users can comment a file, vote it or upload a new one. It is important to note that an upload can be described as a response to another previous uploaded file. Furthermore, the Channel indicates where is the thematic of such file.

Moreover, users can define who are their friends. There exists a subclass of User, the Administrator, which has the power of delete files that can be unsuited for sharing into the system.

This simple model will allow us to explain the expressions and constructions of the Object Constraint Language 2.0 with examples over it. If needed, it will be completed with extra classes, attributes, operations or relations.

6.3 OCL 2.0 EXPRESSIONS

Along this section our aim is to introduce the different elements that can be used inside OCL 2.0 expressions. First of all we will describe the different types and literals that conforms the basis of the language, with a mention to the standard operations over them.

Furthermore, we show the concept of iterator expression and the possibilities of navigation through the different elements of an UML model.

6.3.1 BASIC TYPES

Our first step in the learning phase of the OCL 2.0 language is the explanation of the basic types that will be used as constants within the expressions.

These types are basic data types and are well-known concepts for those that have experience with programming languages.

6.3.1.1 Integer

This is a numbered type, whose instances are all the numbers that can be written without fractional or decimal component, including both their negatives and 0. Examples of Integer numbers can be shown below.

… -3, -2, -1, 0, 1, 2, … , 256, 257, 258, … , 10234, …, 45674, … and so on

Example 6.1: Integers

It is important to note that in the OCL 2.0 language, the Integers are not written separating thousands by neither dot nor comma symbols. All the digits are written preceding the previous one without any character between them.

6.3.1.2 Real

It is another numbered type, whose instances are all the numbers that exist, and therefore the Integers are a subset of them.

There are different notations to use in order to write Real numbers. Furthermore, to separate the decimal component the dot sign is used.

At next examples we will show all the three different notations.

1) … -3, -2, -1, 0, 1, 2, … , 256, 257, 258, … , 10234, … like the Integers.

2) 23.56, 45.0001, 0.00001, -0.9873 using the dot '.' to separate the decimal component from the integer one.

3) 3.6e45, 3.6e+45, 3.6e-45, 3.6E45, 3.6E+45, 3.6E-45 using the exponential notation for big or litte numbers.

Example 6.2: Reals

6.3.1.3 String

This basic type is used to represent arrays of characters, like words. The notation for using Strings within OCL 2.0 expressions is very simple. It is only necessary to brace the alphanumerical characters representing each array of characters with single quotation marks.

`'This is a large String'` ***like the next one***

`'Numbers are allowed'` ***like in*** `'3456'`

`''` ***denotes the empty String.***

Remember to use single quotation marks `'like here'` ***instead of double ones*** `"like here"`***. Double quotation marks are not allowed.***

Example 6.3: Strings

A useful example of how to write Strings in the OCL 2.0 is shown above.

6.3.1.4 Boolean

Our last basic type is the Boolean type. It is useful to denote if something is true or not. Its instances are only two keywords: true and false.

Boolean values: `true` ***(positive, or 1) and*** `false` ***(negative, or 0)***

Example 6.4: Boolean values

As itself indicates, true represents a positive affirmation and false a negative affirmation. They are useful in conditions and comparisons, or as members in logical operations.

6.3.2 Basic Operations

At this point, we are going to show the operations that can be used with the basic types explained before.

6.3.2.1 Integer and Real operations

The following operations are the ones that work with numbers as much Integers as Reals.

Operation	Explanation	Usage
+	Results in the addition value of two numbers. If one of these numbers is Real, the result of applying this operation is also Real.	3 + 2 ⇒ Integer 3 + 0.9 ⇒ Real
–	Results in the subtraction value of two numbers. If one of these numbers is Real, the result of applying this operation is also Real.	4 – 1 ⇒ Integer 5 – 7.9 ⇒ Real
*	Results in the multiplication value of two numbers. If one of these numbers is Real, the result of applying this operation is also Real.	5 * 9 ⇒ Integer 3.5 * 3 ⇒ Real

Operation	Explanation	Usage
–	Unary operation that changes the symbol of the number that precedes. The result type of applying this operation is the same of its source data.	–3 ⇒ Integer –2.56 ⇒ Real
/	Binary operation that results in the value of the first operand divided by the value of the second one. It can be used with both Integer and Real numbers. However, the result is always a Real number.	3 / 2 ⇒ Real 4.7 / 6 ⇒ Real
div	Similar than / division but only for Integer. It cannot be used with Real numbers, and the result of applying it is always Integer.	6.div(5) ⇒ Integer 4.div(2) ⇒ Integer
abs	Unary operation that returns the absolute value of the source data. This operation can be used with Integer or Real numbers.	1.abs() ⇒ Integer (-5.7).abs() ⇒ Real
mod	Binary operation that results in the remainder of dividing the first number by the one between parentheses. This operation cannot be used with Real numbers, and its result is always Integer.	6.mod(4) ⇒ Integer 2.mod(3) ⇒ Integer
floor	Real unary operation that results in the largest integer that is less or equal to the source operand.	(5.2).floor() ⇒ Real (0.9).floor() ⇒ Real
round	Real unary operation that results in the Integer value that is closest to the source operand.	(3.4).round()⇒Real (-2.3).round()⇒Real
max	Binary operation that results in the maximum of the two Integer or Real operands.	4.max(4.5)⇒Integer (1.2).max(0)⇒ Real
min	Binary operation that results in the minimum of the two Integer or Real operands.	5.min(3)⇒ Integer (9.9).min(4)⇒Real
<	Binary operation that returns true if the first operand is less than the second one.	4.5 < 4 ⇒ Boolean 2 < 8 ⇒ Boolean
>	Binary operation that returns true if the first operand is greater than the second one.	6 > 4.4 ⇒ Boolean 16 > 15 ⇒ Boolean

Operation	Explanation	Usage
<=	Relational binary operation that results in a Boolean value according to if the first operand is less or equal than the second one.	4.5 <= 4 ⇒ Boolean 2 <= 8 ⇒ Boolean
>=	Relational binary operation that results in a Boolean value according to if the first operand is greater or equal than the second one.	6 >= 4.4 ⇒ Boolean 16 >= 15 ⇒ Boolean
=	Relational binary operation that results in a Boolean value according to if the first operand is equal than the second one.	56 = 4 ⇒ Boolean 16 = 16 ⇒ Boolean
<>	Relational binary operation that results in a Boolean value according to if the first operand is distinct than the second one.	4.5 <> 4.4 ⇒ Boolean 1 <> 1 ⇒ Boolean

Table 6.1: Integer and Real operations

It is important to note that there are operations that can be used with both Integer and Real numbers because Integers are a subset of Reals, as we explained before.

6.3.2.2 String operations

At this point, the String operations are introduced into Table 6.2.

Operation	Explanation	Usage
=	Relational binary operation that results in a Boolean value according to if the first operand is equal than the second one.	'hello'='bye'⇒Boolean 'John'='John'⇒Boolean
<>	Relational binary operation that results in a Boolean value according to if the first operand is distinct than the second one.	'3d'<>'4d'⇒Boolean 'Joe'<>'Joe'⇒Boolean
size	Results in the Integer value indicating the number of characters of the source operand.	'Maria'.size()⇒Integer ' '.size() ⇒ Integer (0)
concat	Results in the concatenation String of the two operands present in this binary operation.	'a'.concat('b') ⇒ String 'U.S'.concat('.A')⇒String

Operation	Explanation	Usage
substring	Unary operation that results in the sub-String of the source String starting from start index to the end index. Both indexs must be Integer numbers.	'Linux'.substring(3,4) ⇒ 'nu' String
toInteger	Converts the source String into an Integer number if possible.	'28'.toInteger()⇒Integer
toReal	Converts the source String into a Real number if possible.	'7.9'.toReal()⇒Real

Table 6.2: String operations

It is important to remember that String constants are always written between single quotation marks ' ' but not with double quotation marks " ".

6.3.2.3 Boolean operations

To conclude the basic operations, we introduce the operations that work with Boolean values. Remember that these values can be only true or false.

Operation	Explanation	Usage

=	Relational binary operation that results in a Boolean value according to if the first Boolean operand is equal than the second one.	true = true
		true = false

<>	Relational binary operation that results in a Boolean value according to if the first Boolean operand is distinct than the second one.	false <> true true <> true
or	Logical binary operation that results in the Boolean value true according to if at least one of the two operands is true.	true or true false or true
xor	Logical binary operation that results in the Boolean value true according to if either one of the two operands is true, but not both.	true xor true false xor true
and	Logical binary operation that results in the Boolean value true according to if both of the two operands are true.	true and true true and false

Operation	Explanation	Usage
not	Logical unary operation that results in the Boolean value opposite than the source one	not true not false
implies	Logical binary operation than returns Boolean value true if first operand is false or if it is true and the second one is also true.	true implies true false implies true

Table 6.3: Boolean operations

So, inside these three previous tables all the basic operations and their explanation and usage are contained. Such operations are well known for those that have experience in mathematics, logic or programming languages.

6.3.3 COLLECTIONS

Collections are structured data types that allow encapsulating more than one element of a same type inside.

There are four kinds of collections in the OCL 2.0 language: Set, Bag, OrderedSet, and Sequence. A Set is a container where each element inside appears only one time. Therefore, it does not contain duplicate elements. On the other hand, a Bag is like a Set but with duplications allowed.

Moreover, OrderedSet and Sequence are the same as Set and Bag in which the elements are ordered.

All of these constructions can be used as literals inside OCL 2.0 expressions in the same way we can use Integers, Reals, Strings or Booleans within them. Therefore, to know the explicit notation to use with such constructions is mandatory in order to advance a step in the comprehension of the OCL 2.0 language.

6.3.3.1 Collection literals

At next example we can see how to use collections as literals and which are the main differences between the four kinds of collections.

Set Literals:

`Set{5, 4, 9, 6, 1, 0}` ***is a Set of Integer.***

`Set{6.5, 99.34, 6123.45, 0.001, 45e+12}` ***is a Set of Real.***

`Set{'this', 'is', 'a', 'set'}` ***is a Set of String.***

`Set{true, false}` ***is a Set of Boolean.***

`Set{1..10, 30..(30+2)}` ***is the same Set of Integer than*** `Set{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 30, 31, 32}`

Bag Literals:

`Bag{5, 4, 5, 6, 5, 0}` ***is a Bag of Integer.***

`Bag{5.98, 7, 8, 7, 5.98, 4}` ***is a Bag of Real.***

OrderedSet Literals:

`OrderedSet{1, 2, 5, 8, 11, 25, 67}` ***is an OrderedSet of Integer.***

`OrderedSet{'any', 'ball', 'dance', 'eat'}` ***is an OrderedSet of String.***

Sequence Literals:

`Sequence{-3, -3, -2, -1, 0, 1, 1, 1}` ***is a Sequence of Integer.***

`Sequence{2.45, 2.46, 2.46, 2.47}` ***is a Sequence of Real.***

`Sequence{'bed', 'class', 'class', 'do'}` ***is a Sequence of String.***

Example 6.5: Collection literals

Note that to construct a collection literal we use the kind name of the collection preceding the inside elements, which are written separated by commas and braced with curly brackets.

As shown before, an OrderedSet or Sequence of String is ordered following the alphabetical order of the characters from left to right.

Furthermore, to define a range we can use two delimiter numbers separated by two dots between them, as shown at last Set literal of the Example 6.5.

6.3.3.2 Collections of collections

In addition, to define a collection inside another collection is allowed. This way we can use a collection literal as an element member of another outer collection. Nevertheless, it is important to note that if we use collections inside collections, we cannot also use primitive literals like Integers, Reals, Strings or Booleans in the same element level than a collection literal.

To clarify this behaviour we present the following example showing what is and what is not allowed working with collections of collections.

`Set{Set{1, 3, 5}, Set{9, 12, 2}}` ***is a valid Set of Sets of Integer.***

`Set{Set{1, 3, 5}, 8, Set{9, 12, 2}}` ***is not a valid Set of Sets of Integer because it mixes collection literals and primitive literals as elements inside a collection. The* 8 *Integer literal cannot be used here.***

A correct version of the previous Set could be as follows:

`Set{Set{1, 3, 5},` **`Set{8}`**`, Set{9, 12, 2}}`

An inverse solution is to avoid using collections of collections like in:

`Set{1, 3, 5, 8, 9, 12, 2}`

Example 6.6: Collections of collections

6.3.4 COLLECTION TYPES

In some OCL 2.0 constructions it is necessary to define a type. Examples of these constructions are the let expressions, the iterate construction or the parameter description inside a new operation defined in a *def* constraint.

As we will study these constructions in next sections, it is important to know how to define a collection type.

Notation: *'CollectionKindName' + '(' + 'ElementType' + ')'*

Examples:

`Set(Integer)` ***is a collection type of a Set of Integer***

`Set(Set(Integer))` ***is a collection type of a Set of Sets of Integer***

`Bag(Set(Real))` ***is a collection type of Bag of Sets of Real***

`OrderedSet(String)` ***is a collection type of OrderedSet of String***

`Sequence(Sequence(Sequence(Real)))` ***is a collection type defining a Sequence of Sequences of Sequences of Real.***

`Set(User)` ***is a collection type defining a Set of User objects. Remember that User is a UML class of our previous example model to constraint and a UML class can be used as a Type.***

Example 6.7: How to define a collection type

In the example above we have shown ways of how to define collection types. It is important to observe that here we use parentheses to brace the type of the element inside a collection type whereas with collection literals we use curly brackets.

6.3.5 Basic Collection Operations

At this section we will introduce the operations that appear in the OCL 2.0 specification as allowed to be applied to collections.

First of all we show the basic operations and then at next section we will explain the concept of iterator over a collection, including its notation and behaviour.

6.3.5.1 General collection operations

All the operations over collections defined here can be applied to any kind of collection, so they can be used with a Set, a Bag, an OrderedSet and a Sequence. For each operation its notation and usage are shown, including a sample description of what it does with the source collection.

In all of the next usage notations in the following tables, a collection will be noted with the **C** character. Furthermore, **T** will denote the type of the elements inside the collection **C**. Finally the word **elm** indicates an object element.

Operation	Explanation	Usage
size	Operation without parameters that returns an Integer representing the number of elements in the source collection.	C->size() returns Integer
includes	It returns true if the parameter is included inside collection C. Such element must have the same type as the elements of the collection.	C->includes(elm:T) returns Boolean
excludes	It returns true if the parameter is not included inside collection C. Such element must have the same type as the elements of the collection.	C->excludes(elm:T) returns Boolean
count	Return the number of times the parameter element occurs in the source collection.	C->count(elm:T) returns Integer
includesAll	Returns true if all the elements in collection C2 are members of the C collection. Both collections must have compatible elements to be correct.	C->includesAll(C2) returns Boolean
excludesAll	Returns true if all the elements in collection C2 are not members of the C collection. Both collections must have compatible elements to be correct	C->excludesAll(C2) returns Boolean
isEmpty	Returns true if the collection has no elements.	C->isEmpty() returns Boolean
notEmpty	Returns true if the collection has elements inside.	C->notEmpty() returns Boolean
sum	Returns the addition of the elements inside the collection. These elements must support the + operation (e.g., Integers or Reals).	C->sum() returns elm:T
product	It is the Cartesian product of C with C2. Returns a Set containing tuples of two elements, where first one belongs to C and second one belongs to C2.	C->product(C2) returns Set(Tuple)

Table 6.4: Collection operations

6.3.5.2 Bag operations

At this point we will introduce the Bag operations. **B** character indicates a Bag here, and **S** a Set.

Operation	Explanation	Usage
=	Returns true if both Bags are equal.	B = B2 returns Boolean
union	This operation can have a Bag or a Set as a parameter. Returns the parameter elements including the source elements in a new Bag.	B->union(B2 or S) returns Bag
intersection	Returns the collection parameter with only the elements that also appear inside the source Bag.	B->intersection(B2 or S) returns Bag or Set
including	Returns the Bag with the source Bag elements plus the parameter.	B->including(elm:T) returns Bag
excluding	Returns the source Bag without the element parameter.	B->excluding(elm:T) returns Bag
count	Returns the number of occurrences of the element in the source Bag.	B->count(elm:T) returns Integer
flatten	If the source Bag is a collection of collections, it results in the Bag containing all the elements of all the elements of it.	B->flatten() returns Bag
asBag	Returns a Bag identical to source one.	B->asBag() returns Bag
asSequence	Returns a Sequence containing all the source elements in undefined order.	B->asSequence() returns Sequence
asSet	Returns the Set containing all the source elements without duplicates.	B->asSet() returns Set
asOrderedSet	Returns an OrderedSet containing all the source elements without duplicates, in undefined order.	B->asOrderedSet() returns OrderedSet

Table 6.5: Bag operations

6.3.5.3 Set operations

In this section we will show the Set operations.

Operation	Explanation	Usage
=	Returns true if both Sets are equal.	S = S2 returns Boolean
union	Returns the parameter including the elements belonging to the source Set. The input parameter can be a Bag or a Set.	S->union(S2 or B) returns Set or Bag
intersection	Returns a Set with only these elements that appear inside both the source Set and the parameter.	S->intersection(S2 or B) returns Set
ñ	Returns a Set with the elements of the source Set, which are not in the parameter.	S->ñ(S2) returns Set
including	Returns the Set with the source Set elements plus the parameter.	S->including(elm:T) returns Set
excluding	Returns the source Set without the element parameter.	S->excluding(elm:T) returns Set
symmetric Difference	Returns the set containing all the elements that are in source Set or S2 parameter, but not in both.	S->symmetricDifference(S2) returns Set
count	Returns the number of occurrences of the element in the source Set.	S->count(elm:T) returns Integer
flatten	If the source Set is a collection of collections, it results in the Set containing all the elements of all the elements of it. If not, the source Set is returned.	S->flatten() returns Set
asSet	Returns a Set identical to source one.	S->asSet() returns Set
asBag	Returns a Bag identical to source Set.	S->asBag() returns Bag

Operation	Explanation	Usage
asSequence	Returns a Sequence containing all the source elements in undefined order.	S->asSequence() returns Sequence
asOrderedSet	Returns an OrderedSet containing all the source elements without duplicates, in undefined order.	S->asOrderedSet() returns OrderedSet()

Table 6.6: Set operations

Note that when an operation has a parameter, it has to be compatible to the source Set or the elements inside it, according to whether the parameter is a collection or an element.

6.3.5.4 OrderedSet operations

At this section the OrderedSet operations are shown. The character **O** identifies an OrderedSet, and **I** an Integer value.

Operation	Explanation	Usage
append	Returns the source OrderedSet containing the parameter at last position.	O->append(elm:T) returns OrderedSet
prepend	Returns the source OrderedSet with the parameter at first position.	O->prepend(elm:T) returns OrderedSet
insertAt	Returns the source OrderedSet with the parameter inserted at position i.	O->insertAt(i:I, elm:T) returns OrderedSet
subOrderedSet	Returns an OrderedSet containing the elements of the source OrderedSet placed between i and j positions.	O->subOrderedSet(i:I,j:I) returns OrderedSet
at	Returns the element of the source OrderedSet at position i.	O->at(i:I) returns elm:T
indexOf	Returns the Integer value indicating the position of the parameter inside the source OrderedSet.	O->indexOf(elm:T) returns Integer

Operation	Explanation	Usage
first	Returns the first element of the source OrderedSet.	O->first() returns elm:T
last	Returns the last element of the source OrderedSet.	O->last() returns elm:T

Table 6.7: OrderedSet operations

6.3.5.5 Sequence operations

In this section the Sequence operations are shown. **S** character identifies a Sequence, and **I** an Integer value.

Operation	Explanation	Usage
=	Returns true if both sequences operands are equal.	S = S2 returns Boolean
count	Returns the number of occurrences of the element in the source Sequence.	S->count(elm:T) returns Integer
union	Returns the parameter including the elements belonging to the source Sequence.	S->union(S2) returns Sequence
flatten	If the source Sequence is a collection of collections, it results in the Sequence containing all the elements of all the elements of it. If not, the source Sequence is returned.	S->flatten() returns Sequence
append	Returns the source Sequence containing the parameter at last position.	S->append(elm:T) returns Sequence
prepend	Returns the source Sequence containing the parameter at first position.	S->prepend(elm:T) returns Sequence
insertAt	Returns the source Sequence with the parameter inserted at position i.	S->insertAt(i:I,elm:T) returns Sequence
subSequence	Returns a Sequence containing the elements of the source Sequence placed between i and j positions.	S->subSequence(i:I,j:I) returns Sequence

Operation	Explanation	Usage
at	Returns the element of the source Sequence at position i.	S->at(i:I) returns elm:T
indexOf	Returns the Integer value indicating the position of the parameter inside the source Sequence.	S->indexOf(elm:T) returns Integer
first	Returns the first element of the source Sequence.	S->first() returns elm:T
last	Returns the last element of the source Sequence.	S->last() returns elm:T
including	Returns the Sequence with the source Sequence elements plus the parameter.	S->including(elm:T) returns Sequence
excluding	Returns the Sequence without the source Sequence elements plus the parameter.	S->excluding(elm:T) returns Sequence
asBag	Returns a Bag containing all the source elements.	S->asBag() returns Bag
asSequence	Returns a Sequence like the source one.	S->asSequence() returns Sequence
asSet	Returns a Set containing all the source elements without duplicates.	S->asSet() returns Set
asOrderedSet	Returns an OrderedSet containing all the source elements without duplicates.	S->asOrderedSet returns OrderedSet

Table 6.8: Sequence operations

Once we have studied all the collection operations it is important to note that such operations use the arrow operator (->) to link the source collection with the operation header.

Although the distinct (<>) operation is not specified in the OCL 2.0 specification for collections, we think that to use it is not a mistake. Therefore, we assume that all the collection kinds can be applied as operands of such relational operation.

6.3.6 ITERATORS

Once we know how to use collection as literals and which are the operations that can be used with them, the next step is to study another special constructions that also have a collection as source: the iterators.

These constructions provides a powerful way of create new collections from existing ones. All the inner elements of a collection are scanned and in many cases all of them that fulfill a condition are returned.

The notation of the iterator expressions can be written in three ways of complexity. At next example we can see how to write them properly.

Simple notation:

```
collection->iteratorName(expression)
```

Notation with iterator variables:

```
collection->iteratorName(v1,..,vn|expression-with-v's)
```

Notation with typed iterator variables:

```
collection->iteratorName(v1,..,vn:Type|expression-with-v's)
```

Example 6.8: Iterators notation

The notation with iterator variables implies that such variables may appear into the body expressions of the iterator. Furthermore, the type of these variables must be compatible with the type of the elements inside the source collection.

If the type is not specified, it will be used the same type of the inner elements of the source collection. Therefore it is important to know what is the type of such element before using an iterator expression.

At next table we will introduce the different iterators that are specified into the OCL 2.0 specification file that can be found at [Obj06]. The **C** character identifies a collection here.

Iterator	Explanation

SELECT
C->select(boolean-exp)
C->select(v|boolean-exp-with-v)
C->select(v:Type|boolean-exp-with-v)
	Returns a subcollection of the source collection containing all elements for which boolean-exp is true

REJECT
C->reject(boolean-exp)
C->reject(v|boolean-exp-with-v)
C->reject(v:Type|boolean-exp-with-v)
	Returns a subcollection of the source collection containing all elements for which boolean-exp is false.

COLLECT
C->collect(exp)
C->collect(v|exp)
C->collect(v:Type|exp)
	Returns the collection of elements that results from applying exp to every element of the source collection. If such elements are already collections, it just returns their inner elements but not the collection at all.

COLLECT NESTED
C->collectNested(exp)
C->collectNested(v|exp-with-v)
C->collectNested(v:Type|exp-with-v)
	Returns the collection of elements that results from applying exp to every element of the source collection.

FOR ALL
C->forAll(boolean-exp)
C->forAll(v|boolean-exp-with-v)
C->forAll(v:Type|boolean-exp-with-v)
	Returns true if the boolean-exp evaluates to true for each element in the source collection.

EXISTS
C->exists(boolean-exp)
C->exists(v|boolean-exp-with-v)
C->exists(v:Type|boolean-exp-with-v)
	Returns true if boolean-exp evaluates to true for at least one element in the source collection.

IS UNIQUE
C->isUnique(exp)
C->isUnique(v|exp-with-v)
C->isUnique(v:Type|exp-with-v)
	Returns true if exp evaluates to a different value for each element in the source collection

ANY
C->any(boolean-exp)
C->any(v|boolean-exp-with-v)
C->any(v:Type|boolean-exp-with-v)
	Returns any element in the source collection for which boolean-exp evaluates to true.

Iterator	Explanation

ONE
C->one(boolean-exp)
C->one(v|boolean-exp-with-v)
C->one(v:Type|boolean-exp-with-v)
	Returns true if there is exactly one element in the source collection for which boolean-exp is true.

SORTED BY
C->sortedBy(exp)
C->sortedBy(v|exp-with-v)
C->sortedBy(v:Type|exp-with-v)
	Returns an OrderedSet or a Sequence, depending on if source collection has duplicates, with the same elements than source collection but ordered according to exp.

Table 6.9: Iterators

Even if in the previous table we only have shown iterators with one variable, we can use as much variables as we need.

6.3.7 Tuples

Another structured data type similar to collections are the Tuples. These are also containers of other data elements but in this case such elements can have different types.

Each element belonging to a tuple is identified by its name and has a type and a value. At next example we will show an example of tuple literal in order to understand their notation.

```
Tuple{number:Integer=0, word:String='my String'}
```

is a tuple with two fields, number and word. First one is an Integer, which value is 0, and second one is a String with 'my String' as its value.

Therefore, to create a tuple literal we begin with the keyword Tuple followed with an expression between curly brackets. Inside such expression each field of the tuple appears with its name (required) and type (optional) separated for a colon and preceding its value (required) with an equal sign. Each field description is separated with commas.

Example 6.9: Tuple literals

It is important to note that the way of access to a field of a tuple literal is using the dot '.' operator. At following example we show some accesses to tuple fields.

```
Tuple{number:Integer=0, word:String='my String'}.number = 0
Tuple{a:String='a', b:Integer=10, c:Real=0.1}.c = 0.1
Tuple{b:Set(Integer)=Set{1..10}, d:Real=1.1}.b =
Set{1,2,3,4,5,6,7,8,9,10}
```

With these examples we can see different ways of define a tuple literal and access one of its fields. The third example is interesting because it declares a Set as its first field.

Example 6.10: Access to tuple fields

6.3.8 Tuple Types

In the same way we explained how to declare a collection type, at this section we will introduce the concept of tuple type and its notation into the OCL 2.0 language.

At next example we show the notation, which is similar than such that was used to define collection types.

`TupleType(a:Integer, b:Real, c:String, d:Boolean)` ***is a tuple type with four fields. It is important to note that both the name and the type of each field are required. In this case, the value of each field cannot be used here.***

```
TupleType(a:TupleType(a1:Integer,a2:Real),b:Set(Real))
```

We also can define tuple types as a field type inside a tuple type as in the previous example.

Be carefully with the collection and tuple type declaration because they use common parentheses instead of curly brackets, which are only used when defining literals.

Example 6.11: Tuple types

Once we have seen how to define a tuple type, we can use it to define tuple literals as fields of another outer tuple literal, as in next example:

```
Tuple{a:TupleType(a1:Real,a2:Real)=Tuple{a1:Real=0.2,
a2:Real=0.3},b:Real=0.1}.a.a1 = 0.2
```

To define a tuple literal inside another tuple literal as a field, we have to do the same than explained at Example 6.9. So we can access to this inner tuple literal by its name, as in the previous example. In this case we use 'a' to acces the first field, and then we access to 'a1' attribute of this inner tuple literal resulting in the 0.2 Real.

Example 6.12: Tuple literals inside tuple literals

6.3.9 Navigations

Working with UML model elements is a basic task when using the OCL 2.0 language to create constraints. Therefore, to know how to access to different elements of a model from one specific location is mandatory. These accesses are known as navigations through the model.

If we remember the schema presented in Figure 6.1 at the beginning of this chapter, we can find classes, attributes, operations and relationships between classes.

At next subsections we will explain how to navigate over them in an OCL 2.0 expression. This will be useful to make complex constraints in an easiest way.

6.3.9.1 Attributes

First of all we will be centred into the concept of attribute and the access to it from an UML class.

We can define an attribute as a class member field, which is identified by a name and a type. For example, in our model we have the UML class User that has four attributes: nickname and password Strings, an Integer named age, and another String representing a mail direction.

In the OCL 2.0 language, if we want to apply a constraint or rule to an attribute inside a Class we must fix such class as current context, and then to access the attribute by its own name.

But if we are in the middle of an expression that has a UML class as a result type, we can access its attributes like we did in tuple literals. At next example we can see how it works.

Imagine that p is a Set of User instances and we want to apply the forAll operation to it:

```
p->forAll(u1,u2:User|u1.nickname = u2.nickname
                                          implies u1 = u2)
```

As you can see, we have defined two iterator variables inside the forAll iterator, and both variables have the same type than the source collection elements. In this case, it is a Set of User, so both u1 and u2 have User as type, and User is an UML class.

Therefore, as we explained before, the body expression of a forAll iterator must be Boolean. We access to the nickname attribute of User class by means of the dot '.' notation as in tuple literal case.

In this body expression we compare the nickname attributes of two instances of User. This comparison can be made because such attributes are Strings and the equality operation is defined for this data type. With the implies operation we have indicated that if both instances have the same nickname they are the same instance.

This is a way of note that nickname attribute is the identifier of the UML class User and must be different for all its instances.

Example 6.13: Access to attribute of class

It is important to emphasize that with the dot '.' notation we can access to both attributes of class and fields of tuple literals.

6.3.9.2 Association ends and association classes

Another element of an UML model is the association. To navigate over an association we should use the role name of the opposite end in the association with the dot '.' notation.

If such role name is not specified, the class name in the opposite end must be used instead. Furthermore, when the association has more than two ends (e.g., ternary or quaternary associations) to navigate from one end to another it is mandatory to pass through the association class representing such association. If the association class does not appears the name of the relation should be used instead.

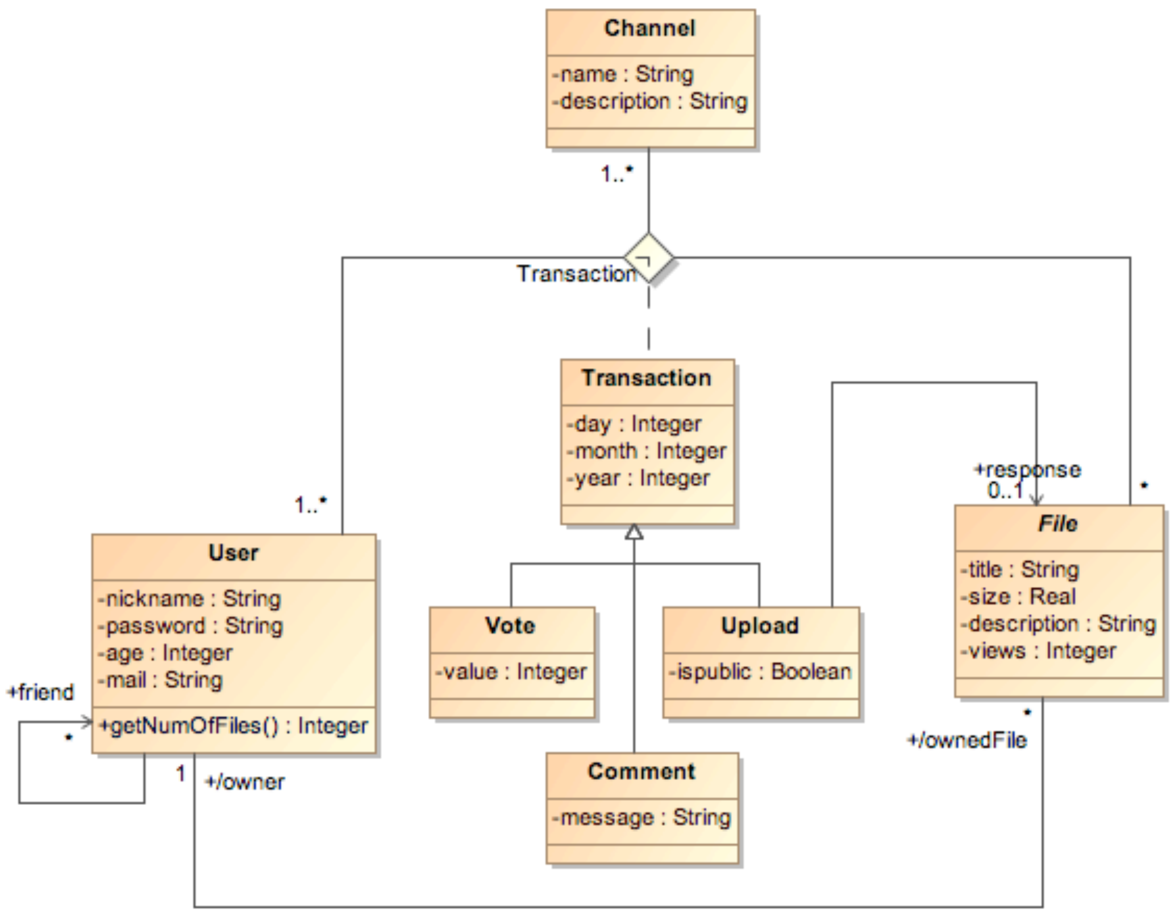


Figure 6.2: Associations

At next example we will show how to navigate through associations.

> ***If we have a variable u, which type is User class:***
>
> `u.ownedFile` ***navigates from User to File classes and results in a Set of File.***
>
> `u.friend` ***navigates from User to User class and results in a Set of User.***
>
> `u.file` ***is not a valid navigation because exists and associative class and the association has more than two members. In this case to go through such class is mandatory to navigate to the opposite end.***
>
> ***A solution for this case should be:*** `u.transaction.file` ***which results in a Bag of File. It is important to note that the name of the classes in a navigation must be used with the first character in lower-case.***

Example 6.14: Navigate through associations

The result of navigation through an association depends on the multiplicity of the association end. If such multiplicity is 1, the result is an instance of the target class. On the other hand, if the multiplicity is greater than 1 it results in a Set.

Moreover, the navigations through associations can be chained, so in this case if we have a Set and navigate to an association end that has multiplicity greater than 1, the result will be a Bag.

If we are in an associative class and navigate to one of its ends, it will result in a single instance of the end class. And if we navigate to an associative class, the result is a Set of instances of such associative class.

All of these results of navigations have some complexity. A complete explanation can be found in [Obj06]. If during some constraint construction or next example it was necessary to explain in more detail this behaviour, it will be clarified there because the previous description should be sufficient for to understand the basis of the navigations.

6.3.9.3 Class operations

Finally, we can use operations of classes in our OCL 2.0 expressions. The way of use them is easy. We only have to follow the same instructions than with a class attribute, but indicating the parameters of the operations.

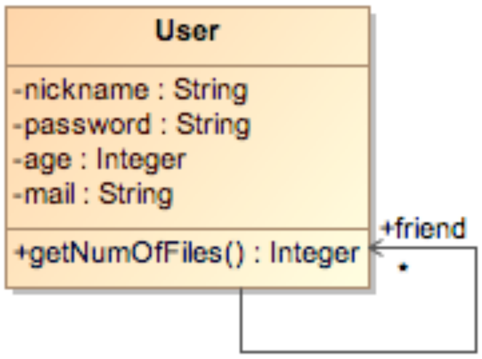


Figure 6.3: User class

In the User class of our model we find the *getNumOfFiles()* operation. At next example we will see how to call this operation.

Our operation to call has the next header:

```
getNumOfFiles(): Integer
```

So, imagine that u is a variable with User as a type. To call this operation we can do as follows:

```
u.getNumOfFiles() <= 100
```

With this constraint we limit the maximum number of files per user.

Example 6.15: Navigation to class operations

If the operation needs any parameter, we only have to insert a literal expression in the same place where the parameter is located. At next example we will show a simple case where a parameter is needed.

If our operation to call has the next header:

```
getNumOfFiles(String name): Integer
```

So, imagine that u is a variable with User as a type. To call this operation we can do as follows:

```
u.getNumOfFiles('JohnSmith') <= 100
```

With this constraint we only limit the maximum number of files of the indicated user.

Example 6.16: Operation call with parameters

6.4 CONSTRAINTS CONSTRUCTION

At previous sections we have shown different basic constructions to make OCL 2.0 expressions. All of these must be placed inside a specific context. This is the moment to introduce the concept of context and the different constraints and rules that one can make with the OCL 2.0 language over an UML model.

6.4.1 CLASSES

In the context of an UML class, the OCL 2.0 language provides the possibility of state invariants and to make definitions of new attributes or query operations.

6.4.1.1 Invariants

An invariant is a boolean rule that always must be true. At next definition we show the correct notation for an OCL 2.0 invariant context.

```
context Classname inv constraintName : boolean-expression
```

Definition 6.1: Invariant context notation

Between the context and inv keywords we have to place the name of the UML class that will be the context of our invariant. As context, such class will become in the starting point for navigations. The constraint name is optional.

It is important to note that an invariant may be written in different contexts. To choose the one that simplifies the expression is part of our job.

Example of invariants over the model in Figure 6.1:

"File size must be less than 10 MBytes"

```
context File inv lessthan10MB : size <= 10.0
```

"A File is identified by its title"

```
context File inv titleID :
  File.allInstances()->forAll(f1,f2:File|f1.title = f2.title
                                                  implies f1 = f2)
```

Note that the application of allInstances() to a class returns a Set containing all the instances of such class.

"The uploaded files of a user are owned by such user"

```
context User inv uploadersAreOwners:
  ownedFile = transaction->select(t:Transaction |
                          t.oclIsTypeOf(Upload)).file->asSet()
```

Note that the application of oclIsTypeOf() to a class returns true if such class is of type specified in the parameter.

In this case we navigate to transaction in order to obtain the Set of Transaction belonging to the context User. Then we select those instances that are Upload instances. And finally we navigate to the association end File in order to obtain the File instances of such transactions, i.e., the uploaded files of the user. It results in a Bag, so we convert it to a Set with the asSet() operation.

We compare this instances with the Set resulted of the navigation to File class through the owning association. They must be equal to fulfill the invariant.

Example 6.17: Example of invariants

6.4.1.2 Definitions

A definition is a construction that creates a new attribute of class or a new query operation. At next definition we will describe the correct notation for an OCL 2.0 definition context.

Attribute definition:

```
context Classname def constraintName :
    attributeName : attributeType = definition-expression
```

Query operation definition:

```
context Classname def constraintName :
    operationName(parameterlist): returnType =
                                        definition-expression
```

Definition 6.2: Definition context notations

It is important to note that the definition expression type must be compatible with the attribute type defined, or with the return type in a query operation case.

At next example we will see different definitions in order to clarify the notation and to understand how to use definitions in our models with the OCL 2.0 language.

Definition of a new attribute 'votations' in File class that shows the number of votations of a file.

```
context File def numOfVotes : votations : Real =
  transaction->select(t:Transaction|t.oclIsKindOf(Vote))
                                              ->size()
```

Definition of a new query operation that returns the transactions of a concrete year.

```
context Transaction
def : anualTransactions(year:Integer): Set(Transaction)
      = Transaction.allInstances()->select(t|t.year=year)
```

Example 6.18: Example of definitions

6.4.2 Attributes

Attributes are other possible contexts for our OCL 2.0 constraints. At this point, when we use the attribute word we talk about both class attributes and association ends.

We can write derivations and initializations for those elements of a UML model. At next subsections we will study in detail such context constructions.

6.4.2.1 Derivations

A derivation allows to describe how is calculated the value of a derived attribute or association end. At next definition we will note how to construct a derivation context.

```
context Classname::attributeOrAssociationEndName : Typename
derive constraintName : derivation-expression
```

Definition 6.3: Derivation context notation

It is important to note that the type of the attribute or association end must be compatible with the derivation expression. Furthermore, such attribute or association end must exist in the class of the context in the UML model.

Imagine that in File class we have a derived String attribute called 'ownerName' representing the name of the owner of such file. Its derivation rule must be as follows:

```
context File::ownerName : String
derive : owner.nickname
```

Example 6.19: Example of derivation

6.4.2.2 Initializations

An initialization allows to describe how is initialized an attribute or association end. At next definition we will show how to construct an initialization context.

```
context Classname::attributeOrAssociationEndName : Typename
init constraintName : initialization-expression
```

Definition 6.4: Initialization context notation

It is important to note that the type of the attribute or association end must be compatible with the derivation expression. Furthermore, such attribute or association end must exist in the class of the context in the UML model.

We want to denote that the initialization value for the attribute views of a File instance is 0

```
context File::views : Integer
init : 0
```

We want to denote that the initialization value for the association end friends of a User instance is an empty Set

```
context User::friend : Set(User)
init : Set{}
```

Example 6.20: Examples of initializations

6.4.3 Operations

Finally, we can use an operation as a context for OCL 2.0 language expressions. It provides the possibility of describe pre and post conditions of an operation. Furthermore we can write expressions showing the body of an operation, that is, describing the tasks that it should do when is called.

At next subsections we will introduce the concepts of preconditions, postconditions and body expressions, showing their notations and some examples.

6.4.3.1 Preconditions

A precondition is an expression that must be true before calling the operation in which it is declared.

```
context Classname::operationName(paramlist) : resultType
pre constraintName : boolean-expression
```

Definition 6.5: Precondition context notation

It is important to note that the header of the operation defined in the precondition context must exist in the UML model that we are using to constraint.

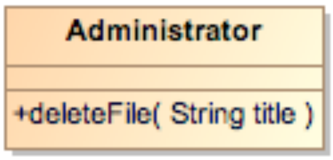


Figure 6.4: Operation declared in an UML class

In our model, we have an operation inside the Administrator class, named deleteFile. If we want to specify a precondition for such operation to note that a File with the title parameter must exist, we should do as follows:

```
context Administrator::deleteFile(title:String)
pre fileExists: File.allInstances()->one(f|f.title = title)
```

Example 6.21: Example of precondition

6.4.3.2 Body of operations

Since the OCL 2.0 language is a declarative language, we can indicate what an operation should do without writing how should do it.

```
context Classname::operationName(paramlist) : resultType
body constraintName : body-expression
```

Definition 6.6: Body expression context notation

A body constraint allows us to show the tasks that are done inside an operation. So, at next example we will show a body constraint for the operation *getNumOfFiles* of User class.

```
context User::getNumOfFiles():Integer
body : ownedFile->size()
```

Example 6.22: Example of body context

It is important to emphasize that we indicate that the number of owned files must be counted, but we do not explicit how to make this compute.

6.4.3.3 Postconditions

A postcondition is an expression that must be true after the call of the operation defined in the postcondition context.

```
context Classname::operationName(paramlist) : resultType
post constraintName : boolean-expression
```

Definition 6.7: Postcondition context notation

We have to remember that a postcondition is a boolean rule, therefore the expression inside a postcondition context must be Boolean.

In the next example we can find a simple postcondition in which the result of it is a boolean value.

If we want to make a postcondition for the deleteFile operation of Administrator class, we should do as follows:

```
context Administrator::deleteFile(title:String)
post fileWasDeleted:
        not File.allInstances()->one(f|f.title = title)
```

Example 6.23: Example of postcondition

6.5 COMPLEX CONSTRUCTIONS

At this section we will introduce other constructions of the OCL 2.0 language that can be useful.

6.5.1 SELF KEYWORD

Within the context of an OCL 2.0 constraint we are able to use a keyword that represents the instance of the contextual UML class: the self keyword. It can be useful when in an expression we have two elements with the same name, but one is an attribute of the contextual class.

```
context User::setNewPassword(password:String)
post newPwd: self.password = password
```

Here the use of self is mandatory to differentiate the password attribute of User from the String parameter, because both have the same name.

To avoid the use of self, we only have to change the name, like in next constraint where self is implicit:

```
context User::setNewPassword(pass:String)
post newPwd: password = pass
```

Example 6.24: Example of self keyword

At example above, we have shown a constraint where self is optional and other one where using self is mandatory.

6.5.2 Comments

In the OCL 2.0 language we have the possibility of writing comments inside the constraints in order to clarify their understandability.

To write a comment we only have to precede each comment line with two hyphens like in the next example:

```
-- This is an OCL comment
context User::getNumOfFiles():Integer
body : ownedFile->size() -- Another comment
-- A final comment
```

Example 6.25: Comments

6.5.3 Multiple contexts

The OCL 2.0 language provides the possibility of reuse a context for more than one constraint. We can write more than one invariant or definition in the same context. Or also it is possible to combine derivations with initializations, and preconditions with postconditions and body expressions.

```
context Classname
 inv: invariant-expression
 def: newattribute : Type = definition-expression
 def: newoperation(paramlist) : Returntype = definition-exp
 inv: another-invariant-expression
```

Example 6.26: Combination of invariants and definitions

At previous example we show this behaviour combining invariants with definitions. At next example we will show how to combine derivations with initializations, and preconditions, postconditions and body expressions.

```
context Classname::attributeOrAssociationEnd:Type
 init: invariant-expression
 derive: derivation-expression

context Classname::operation(paramlist):returnType
 pre: boolean-expression
 pre: another-boolean-expression
 body: body-expression
 post: postcondition-expression
```

Example 6.27: Combination of initializations with derivations, and preconditions with postconditions and body expressions

6.5.4 IF EXPRESSIONS

Like the complete programming languages, the OCL 2.0 provides a condition structure that allows choosing between two alternatives: the if expression.

Such construction evaluates a condition and according to if the result is true or false, it chooses one alternative or the other one.

```
context User
def: adult:Boolean = if age < 18
                      then false
                      else true
                      endif
```

Example 6.28: If expression

It is important to emphasize that both branches inside an if expression are mandatory. In the previous example, the new attribute adult will be true only if the age of the user is greater or equal than 18 years.

6.5.5 Let expressions

A let expression is a construction that allows us to define a temporary variable that can be used inside an expression in order to avoid repetitions or complex rules.

Note that the scope of such variable is only the current context where it is defined.

At next definition we will describe the two accepted notations to write let expressions.

Single notation

```
let variableName:Type = expression
in expression-using-the-previous-variable
```

Multiple notation

```
let varName1:Type = exp1, ..., varNameN:Type = expN
in expression-using-these-variables
```

Definition 6.8: Let notation

At next example the usage of this kind of expression is shown:

```
context File inv onlyOneUploadPerFile :
 let files:Set(File) = File.allInstances()
 in files-> forAll(f:File|f.transaction
        ->one(t:Transaction|t.oclIsTypeOf(Upload)))
```

Example 6.29: Let expression example

6.5.6 Iterate

The iterate construction is similar than the iterators, but it is more general. Such construction can be useful to iterate over a collection of elements and return a custom result.

At next definition we can see how to construct an iterate expression.

```
Collection->iterate(variable-list;acc:Type=expression|
  expression-with-variables-and-acc)
```

The acc variable is the accumulator of the result of the iterate. Furthermore, the variable list can be used like in iterators.

Note that the type of the inner expression should be compatible with the type of the accumulator.

Definition 6.9: Iterate notation

At next example we show how to use an iterate expression inside an OCL 2.0 contraint.

```
context File::voters:Set(User)
derive : let votes:Set(Transaction) =
             transaction->select(t|t.oclIsTypeOf(Vote))
         in votes->iterate(v;res:Set(User)=Set{}|
                             res->including(v.user))
```

First of all we define a variable votes including all the Vote transactions from a file. Then we iterate over such variable adding the User of each Vote transaction into the result Set.

Example 6.30: Iterate usage

6.5.7 @PRE KEYWORD

In a postcondition, if we want to reference some UML element in precondition time in order to note its changes once the operation is applied, we can use the @pre keyword after the name of such element.

At next example we will use the @pre in order to access to the value of an attribute at precondition time. Note that @pre is placed always after the name to qualify. It also can be use with operations, placed before the parameters.

```
context User::birthday()
  post: age = age@pre + 1

context Administrator::deleteFile(title:String)
  post: let user:User = File.allInstances@pre() ->
                          select(f|f.title = title).owner
          in user.ownedFile->size() =
             user.ownedFile@pre->size() - 1
```

Example 6.31: @pre usage

6.5.8 Result keyword

In a postcondition we also can use a special variable called result, which type is equal than the result type of the operation of the context. Look at next example how to work with such variable.

It is important to emphasize that result variable can only be used in a postcondition.

```
context User::getNumOfFiles():Integer
  post: result = ownedFile->size()
```

Example 6.32: Result usage

6.5.9 oclIsNew() operation

The *oclIsNew()* operation can only be used in a postcondition and evaluates to true if its source element is created during the execution of the operation, i.e., such element does not exist at precondition time.

This operation is useful in postcondition of creator operations to note that an element is a new instance of a class.

At next example it is shown how to use this operation in the context of an operation that creates a new Administrator.

```
context Administrator::newAdmin(nickname:String)
  post: adm.oclIsNew() and adm.oclIsTypeOf(Administrator)
        and adm.nickname = nickname
```

Example 6.33: oclIsNew() usage

6.5.10 Use of Qualifiers

The last construction that we will explain at this chapter is related with the navigations. In case of navigate through a recursive relation, i.e., a relation whose end classes are the same class, and such relation has an associative class, if we want to go from to the end class to the associative class, we have to define from which end we are navigating.

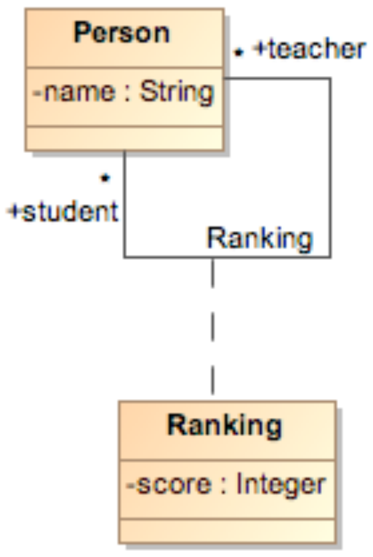


Figure 6.5: Recursive relation

At Figure 6.5 if we want to navigate from Person class to Ranking associative class we have to specify if we come from student or teacher end. This specification is made by means of a qualifier between square brackets.

At next example it is shown how to do this navigation properly.

From Person to Ranking (student branch)

```
context Person inv:
  self.ranking[student].score->sum() >= 0
```

in this case the navigation self.ranking[student] evaluates to the set of Ranking belonging to the collection of students.

From Person to Ranking (teacher branch)

```
context Person inv:
  self.ranking[teacher].score->sum() >= 0
```

in this case the navigation self.ranking[teacher] evaluates to the set of Ranking belonging to the collection of teacher.

From Person to Ranking (unspecified)

```
context Person inv:
  self.ranking.score->sum() >= 0  -- Incorrect usage
```

in this case the unqualified use of the association class name is not allowed in such a recursive situation.

Example 6.34: Qualified navigation through recursive associations

THE OCL 2.0 METAMODEL 7

7 THE OCL 2.0 METAMODEL

7.1 INTRODUCTION

Along this chapter we will explain the OCL 2.0 metamodel defined at [Obj06]. This is an essential piece in the puzzle that represents our OCL expressions processor.

As we explained earlier, the aim of our project is to convert textual constraints to their representation as instances of the OCL 2.0 metamodel. Therefore, to know the different elements that are members of such metamodel is mandatory.

In the following sections we will discuss about the different versions of the OCL metamodel and the one chosen and implemented by our conceptual modeling environment, that is, the Eina GMC tool (see chapter 5).

After that, we will introduce the class diagrams where the different elements and also the relations between such elements of the metamodel are shown. To complete the diagrams, we will note some changes applied into the original OCL 2.0 metamodel in order to fix some problems found along the development phase.

7.2 THE BASIC OCL AND ESSENTIAL OCL

First of all we will note that the programmers of the Eina GMC tool did the development of the OCL 2.0 metamodel. So, we work with this tool like current users once the metamodel was finished, and all the problems found at this stage had to be reported to the Eina GMC staff.

Therefore, one of our purposes was to test the OCL 2.0 metamodel implemented into the Eina GMC environment in order to find possible errors and achieve a more stable version of it.

Analysing the OCL 2.0 metamodel implemented into the Eina GMC tool, we found that its version was a mixture of the abstract version of the complete OCL 2.0 metamodel defined at chapter 8 of the OCL 2.0 Formal Specification (see [Obj06]) and the basic and essential versions introduced at chapter 13 of the same document.

The basic and essential versions of the OCL metamodel contain the minimal OCL elements to work with the common UML metamodel elements: Property, Operation, Parameter, TypedElement, Type, Class, DataType, Enumeration, PrimitiveType, and EnumerationLiteral.

This version was the one implemented initially into Eina GMC environment. Because of our needs of adding more elements to this metamodel in order to support complex OCL 2.0 constructions, a few elements of the complete version were included there.

So, at next section we will show and explain the OCL 2.0 metamodel implemented into the Eina GMC tool, which contains the basic and essential versions completely, and the complete version partially.

7.3 DIAGRAMS

In the OCL 2.0 specification [Obj06], the metamodel is introduced by means of the representation of eight diagrams containing the elements that belong to it. Each diagram denotes a part of the OCL 2.0 constructions and syntax, and together conform the full OCL 2.0 language.

We will comment each diagram and the elements that it contains. Furthermore, we will discuss the changes made to them if necessary and the causes that prompted such changes.

Note that the blue coloured classes are members of the UML 2.0 metamodel. Since OCL is part of the UML, both metamodels have a closer relation and there are elements of the UML metamodel that appear at OCL metamodel.

7.3.1 Types

The OCL 2.0 language is a typed language, so the whole set of elements that conform such language have a type that indicates the result type of their evaluation.

A simple example of it could be seen when we use integer constants into our constraints. As we will see when we explain the diagram of literals, an integer constant is defined in the OCL metamodel as an IntegerLiteralExp, which indirectly is a subclass of TypedElement metamodel class from UML metamodel.

Therefore, IntegerLiteralExp is an element with type, and such type must be a DataType with 'Integer' as name.

In the types diagram shown at Figure 7.1 we found all the classes that represent the different types that an OCL element can have.

We must emphasize the CollectionType class and its subclasses. They represent the different kinds of collection types that could appear as a result of the elements into an OCL constraint. OrderedSetType, SequenceType, BagType and SetType are such kinds, according to the collection types shown at chapter 6.

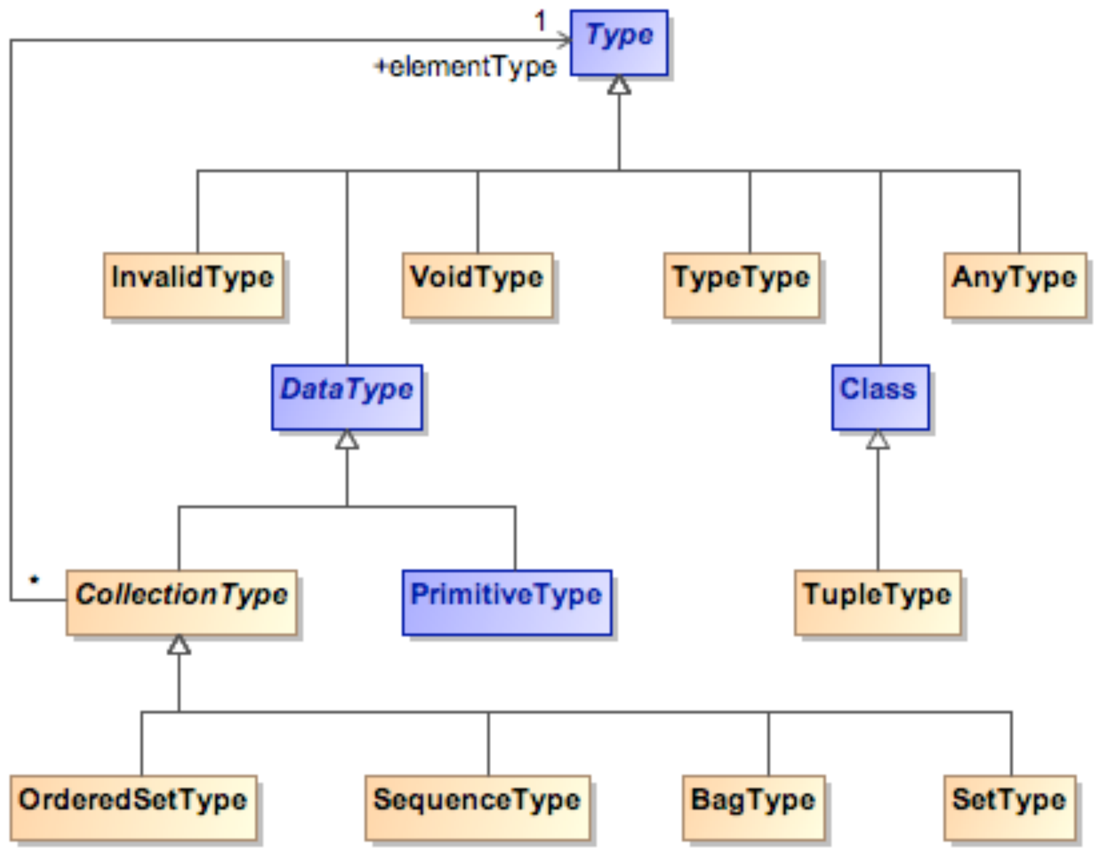


Figure 7.1: Types

Note that to represent the type of the elements inside a collection, the OCL 2.0 metamodel provides a relation between CollectionType class and Type class that allows the collection types to indicate the respective type of their elements.

Another important concept is the DataType. As we can see, it is represented as an abstract class. We decide to maintain this notation because it appears as abstract into the OCL 2.0 metamodel shown at [Obj06], but it is not an abstract class into the UML 2.0 metamodel.

In the implementation of the UML and OCL metamodels into the Eina GMC environment, the DataType is a concrete class as indicated in the UML 2.0

metamodel found at [Obj07]. Therefore, we use this class to represent the four basic types of the OCL 2.0: Integer, Real, Boolean and String.

We either cannot forget the tuples. A tuple type could be represented by different ways depending on what version of the OCL metamodel we choose. In the basic and essential versions, it is indicated that a tuple type is a subclass of Class class of the UML metamodel. Therefore, the members of a tuple are represented by its Property attributes, as a common UML class definition.

In this version, a tuple type (TupleType in the metamodel) also is a subclass of DataType. This double inheritance, that is, subclass of both Class and DataType, was a problem in the implementation of the TupleType in the Eina GMC metamodel. It was caused because DataType also inherits from Class, so we have two ways of extend the internal operations of Class, i.e., extending the operations already extended in DataType or extending directly from Class. So, it caused errors in the metadata repository and we chose to extend TupleType directly from Class and avoid the inheritance of DataType. That is shown in the types diagram.

On the other hand, in the complete OCL version is indicated that TupleType inherits only from DataType. In our alternative we chose to inherit only from Class because we only need the characteristics of a UML class to represent a tuple type and its members, and to inherit from DataType does not provide any additional feature. So it was the reason of our decision.

Finally, we have to note that a Class representing an UML class is also a valid type for an element, as we can see at the previous diagram.

The other classes in the types diagram represent minority types that will be explained if necessary along this document. All of them are implemented as shown in the Figure 7.1 into the OCL 2.0 metamodel of the Eina GMC environment.

7.3.2 The top container expression

In this diagram we show the part of the OCL 2.0 metamodel that has the responsibility of be the wrapper of the OCL expression in a constraint.

As we explained earlier in chapter 3 where we introduce a subset of the UML metamodel, a Constraint metamodel class has an attribute that contains the specification of an OCL expression. This expression is the whole tree representing the textual constraint converted into metamodel instances and it is defined as the ExpressionInOcl metamodel class shown at Figure 7.2.

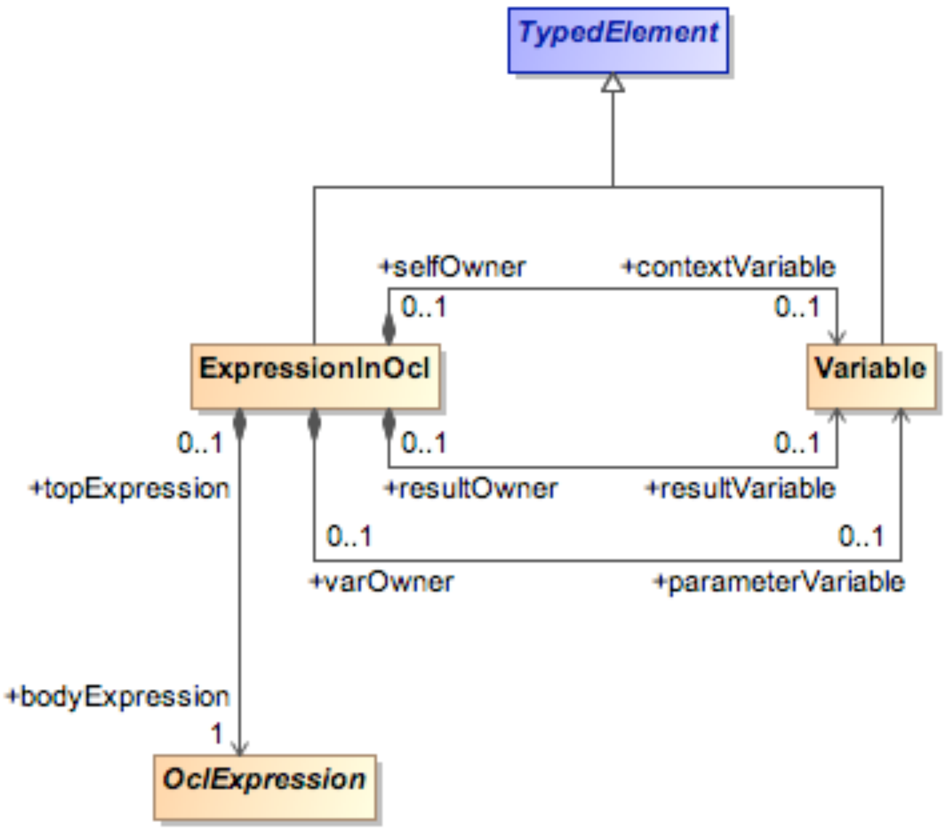


Figure 7.2: Top container expression

Note that ExpressionInOcl has a relation with OclExpression class, which is the superclass of all the expression constructions that can be used in the OCL 2.0 language.

ExpressionInOcl also contains a Variable that represents the context of the OCL 2.0 constraint. This Variable is used into the body OclExpression. It also contains another Variable as the result of the interpretation of the constraint, but we do not use it because the aim of our project is to evaluate and convert textual constraints into metamodel instances, but not to interpret these constraints according to a set of instances of a UML model and to return the result of this interpretation.

It is important to emphasize that both ExpressionInOcl and Variable are subclasses of TypedElement metamodel class. Therfore, they must have a Type indicating their evaluation. The type of and ExpressionInOcl should be the type of its body expression.

On the other hand, the type of a Variable class indicates what element represents this Variable. As an example, a Variable that has a Type, which is a Class representing a UML class of a UML model, could be used in the same way a UML class can be used in a OCL expression. As we will explain after that, these Variable classes are used as iterator variables or with let expressions.

7.3.3 MAIN EXPRESSION CONCEPT

In this section we introduce the different subclasses of OclExpression that represent the distinct constructions that the OCL2.0 provides in order to make constraints.

At Figure 7.3 it is possible to find the LiteralExp and the IfExp metamodel classes. Both metamodel elements will be explained in next sections with their own metamodel diagram. Despite it, these classes are shown here in order to note that are part of the OCL 2.0 expressions.

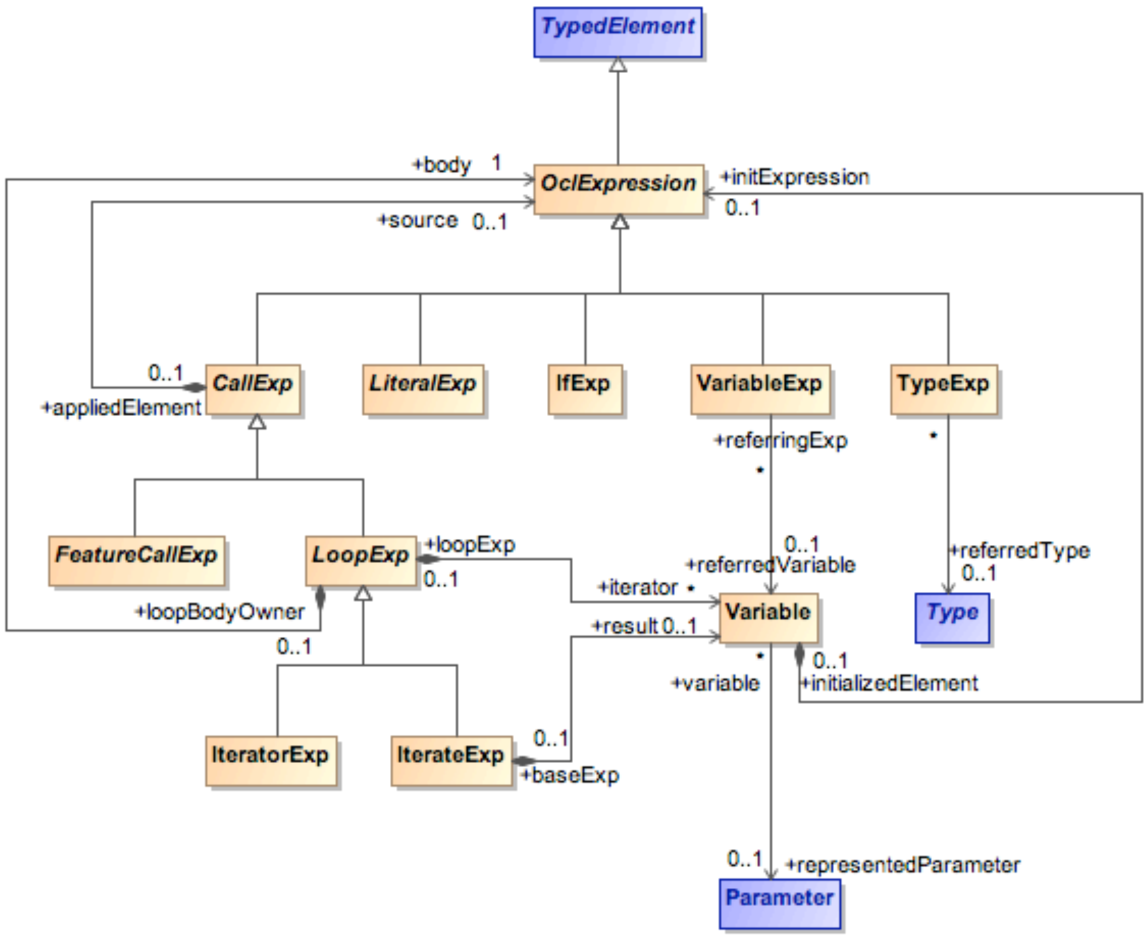


Figure 7.3: Main expression concept

Another metamodel class shown here is the VariableExp that provides the possibility to encapsulate the Variables into an OclExpression. It is useful when we have a Variable as source of a CallExp, which will be explained after that.

Each VariableExp should have a Variable linked as referenced element of this expression. In addition, the Variable class should have an OclExpression as init expression. This can be used in a let expression where a Variable is defined by means of another expression. Once we have converted the let expression into metamodel instances, the init expression is stored in this field of the Variable class.

Similar than VariableExp, we find the TypeExp in this part of the OCL 2.0 metamodel. It is useful to wrap a Type in order to be used as an OCL expression.

As we indicated before, the blue coloured metamodel classes shown at all of these diagrams are part of the UML 2.0 metamodel, but are included here to note the closer relation that exists between these two metamodels.

Finally, the essential expression shown here is the CallExp and its subclasses. This section of the metamodel involves the treatment of operations, iterators, the iterate construction, and even the navigation to attributes, association ends and association classes.

It is important to note that CallExp has a relation with OclExpression representing the source expression that is the element where to apply such CallExp.

The first subclass is FeatureCallExp that contains the navigations and usage of operations. This part will be explained at next section in its own diagram.

We will be centred in the iterators and the iterator construction. Both are LoopExp and have a link with another OclExpression that conforms the body of these constructions. Particularly, this body expression represents the expressions tree included inside both the iterators and the iterator construction.

LoopExp also has a set of Variable objects that represents the iterator variables of the iterators or the iterate construction. These iterator variables are the Variable objects used inside the body expression of such constructions.

Finally, the IterateExp metamodel class representing the iterate construction of the OCL 2.0 language contains another relation with a Variable object defining the common result variable used in this construction.

7.3.4 Feature call expressions

As we introduce at previous version, the FeatureCallExp metamodel class represents both the operation usage and the navigation to attributes, association ends and association classes.

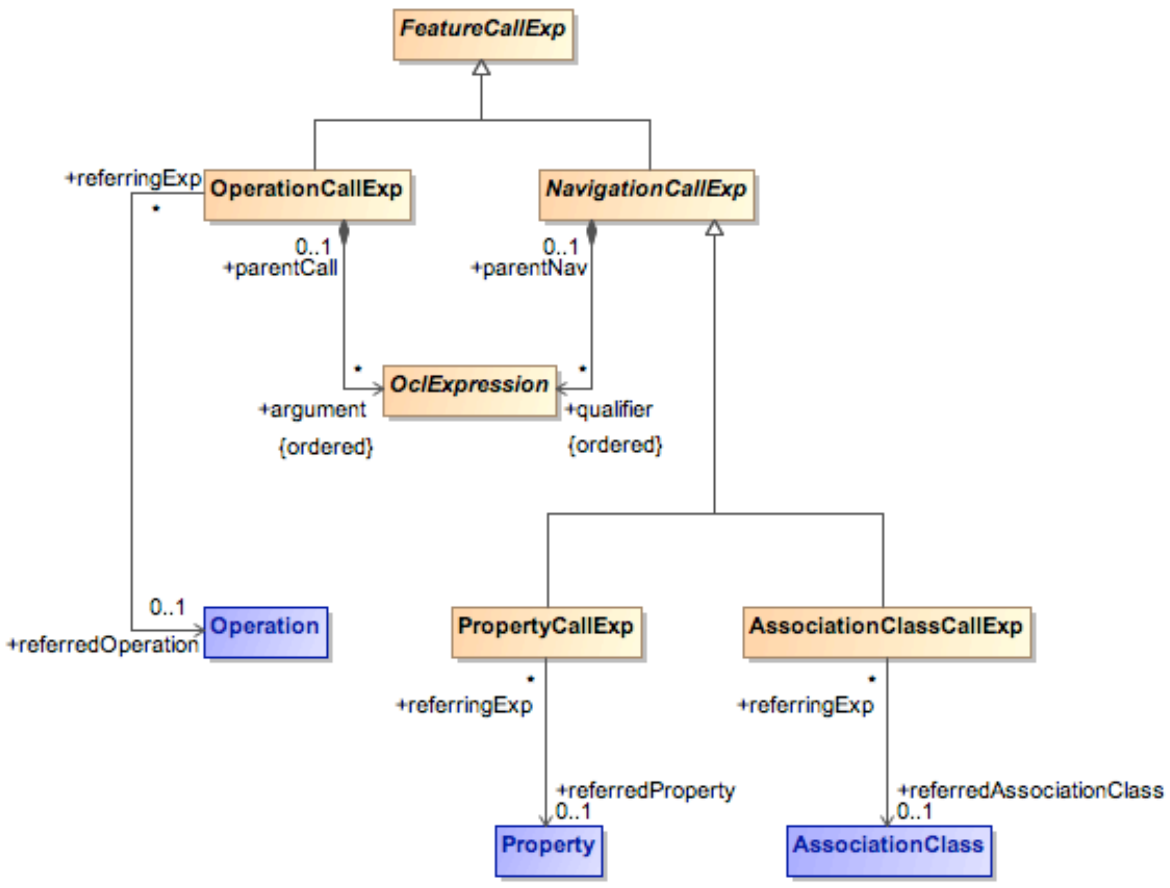


Figure 7.4: Feature call expressions

First of all, the OperationCallExp class defines how to store an operation expression as metamodel instances. Each OperationCallExp has a link with a UML Operation that references the operation used into such expression.

It also contains an optional set of OclExpression that can be used to store the parameters of the operation. However, this set can store the right expression in a binary operation (e.g., the divisor in a division or the right operand of a binary addition).

On the other hand, we have the NavigationCallExp metamodel class to represent the navigation expressions. It has a set of qualifiers, i.e., OclExpression objects, to provide the way of store the qualifying elements in a qualified navigation.

As subclass of NavigationCallExp we have the PropertyCallExp that represents both the call of attributes and the navigation through association ends. This is a change from OCL 1.1 to OCL 2.0 because of the change of the UML in his version 2.0.

In previous versions of the UML, there existed the Attribute and AssociationEnd classes, but now they are joined into the Property metamodel class. So, the AttributeCallExp and AssociationEndCallExp are removed and instead of them we must now use the PropertyCallExp metamodel class.

This change introduces more complexity in the parsing of OCL constraints because one element can be used with different concepts. The PropertyCallExp metamodel class has a link with a Property that can represent an attribute of class or an association end. This complexity will be found in the evaluation process of the type of these PropertyCallExp expressions.

The other subclass of NavigationCallExp is the AssociationClassCallExp metamodel class. Such class is not part of the basic and essential versions of the OCL 2.0 metamodel and we had to add it from the complete OCL in order to support the conversion from textual constraints to metamodel instances of the navigations to association classes.

In the same way of the PropertyCallExp, the AssociationClassCallExp metamodel classes have a reference to the AssociationClass that is the object of the navigation.

7.3.5 If expressions

The If expressions, which are OCL 2.0 constructions, are part of the OCL 2.0 metamodel. The IfExp metamodel class represents such expressions and contains the common parts of a if construction.

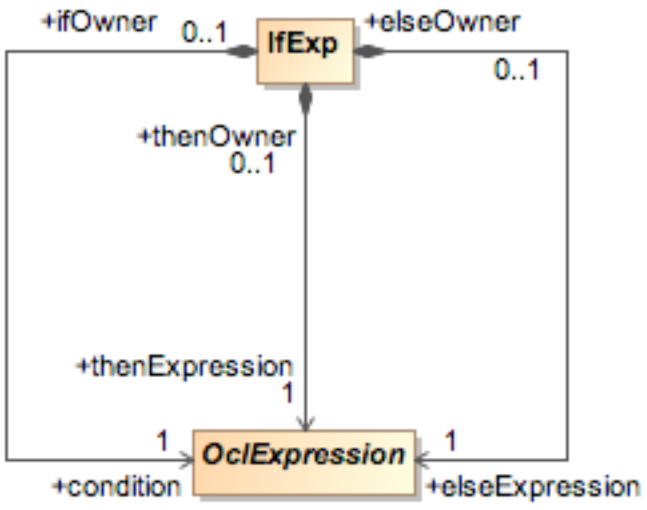


Figure 7.5: If expressions

As we can see in Figure 7.5, there are three OclExpression class references linked to the IfExp. Such expressions store the condition, the then branch and the else branch of an if expression.

Note that as we explained earlier in this document, the then and else branches are mandatory, as indicated with the 1 in the cardinality of the associations from IfExp to OclExpression metamodel class.

Finally, it is important to emphasize that the type of the condition OclExpression will be Boolean if the evaluation of it has no errors.

7.3.6 Let expressions

Let expressions are the last OCL 2.0 constructions to explain. As we explain at previous chapter, a let expression allows defining a variable in order to use it into an expression.

Therefore, the LetExp metamodel class contains a link to an OclExpression representing the expression where the variable can be used, and also contains a Variable reference that is the variable defined into the let expression.

This init expression for such variable is mandatory, so the Variable metamodel class has to contain this reference.

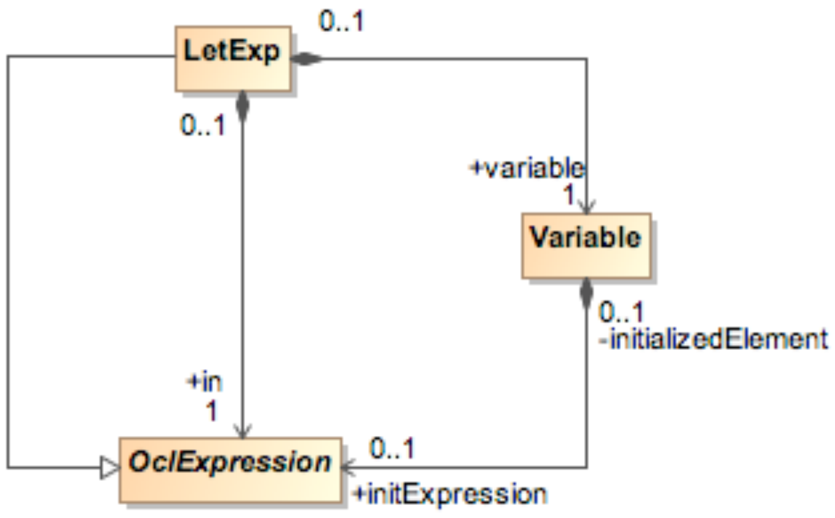


Figure 7.6: Let expressions

Note that if we use the let construction with multiple variable definitions, we have to make a tree of LetExp where the in expression of the outer LetExp will be another LetExp and this behaviour is repeated until all the variables are stored. The last variable is stored in the inner LetExp and its *in* expression will be the expression where to use all the variables.

7.3.7 LITERALS

In this diagram we found the metamodel classes of the OCL 2.0 language representing the literal expressions.

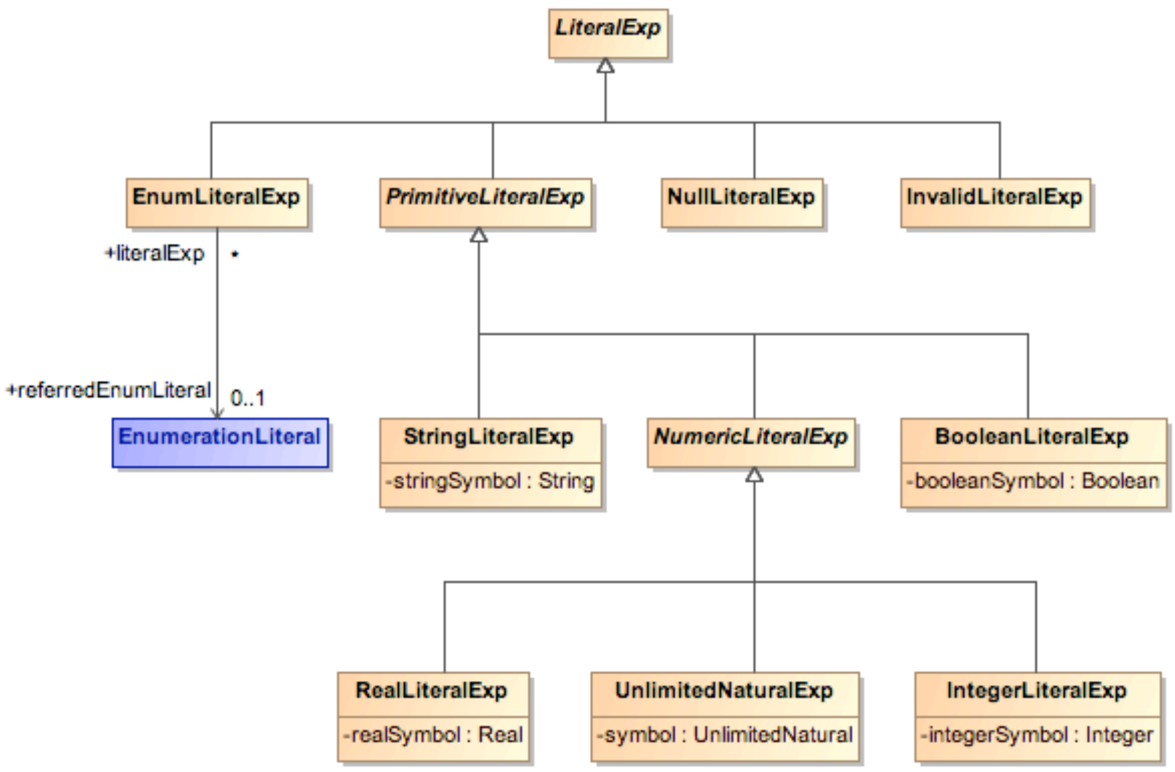


Figure 7.7: Literals

The most important classes here are the EnumLiteralExp and the PrimitiveLiteralExp.

The first represents the usage of an enumeration literal into an OCL 2.0 expression. It contains a reference to the enumeration and works like a wrapper for such enumeration in order to be used like an expression.

On the other hand, PrimitiveLiteralExp is the superclass for the literals that represent the basic types of the OCL 2.0 language, i.e., the Integer, String, Real and Boolean types.

As subclasses of PrimitiveLiteralExp we found StringLiteralExp and BooleanLiteralExp metamodel classes. Both classes have an attribute that is the container of the constant in the textual constraint.

For example, if we use a String into an expression, this String will be converted into a StringLiteralExp, which stringSymbol will be such String. In the same way, the BooleanLiteralExp contains a Boolean attribute to contain the Boolean values used in an expression.

Another subclass of PrimitiveLiteralExp is the NumericLiteralExp class. It class has RealLiteralExp and IntegerLiteralExp as its subclasses. Both of them have a symbol attribute in order to wrap the numeric constant used into an expression. These metamodel classes follow the same behaviour than previous ones.

At Figure 7.7, we can find another literal metamodel classes, like NullLiteralExp, InvalidLiteralExp or UnlimitedNaturalExp. The first two classes are not used in our project because we do not need them since our purpose is to convert textual constraints to metamodel instances and not to interpret the result or to execute these constraints.

In addition to it, the UnlimitedNaturalExp is not used because we decided to use only Integers, so when we use a Natural number we convert it into an IntegereLiteralExp and not into an UnlimitedNaturalExp.

7.3.8 COLLECTION AND TUPLE LITERALS

Our last diagram contains the metamodel representation of another literal expression defining the collection and tuple expressions.

In one side, we have the TupleLiteralExp metamodel class that represents the usage of a tuple as an OCL 2.0 expression. This class contains a set of

TupleLiteralPart objects. Such TupleLiteralPart represents a member of the tuple and has a Property storing the name and the type of such member. In addition, it contains a reference to an OclExpression to store the value of such member in the tuple.

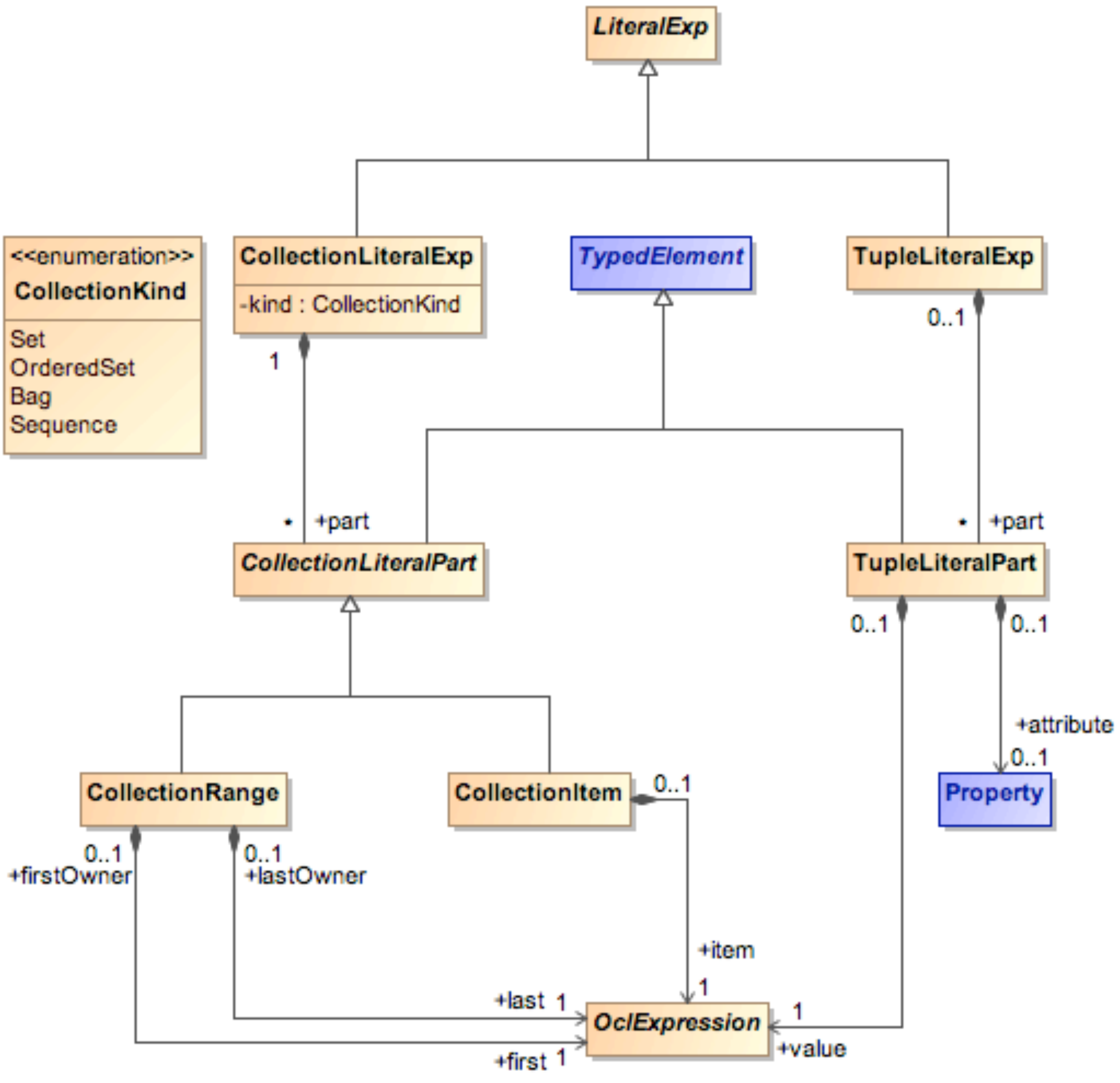


Figure 7.8 Collection and Tuple literals

This value expression does appear neither in the basic and essential OCL versions nor in the complete OCL version, but to store this init value for each member of a tuple is mandatory.

Therefore, to provide this functionality we have decided to add this relation between TupleLiteralPart and OclExpression, indicating that it is mandatory with the 1 cardinality.

Note that the type of a TupleLiteralExp is a TupleType, which was defined at diagram shown at Figure 7.1.

On the other side, we have the CollectionLiteralExp that has an attribute indicating what kind of collection represents. The type of CollectionLiteralExp is a type subclass of CollectionType according to the kind of the collection literal.

Every member of a collection is represented as in a tuple with a set of a metamodel class. In this case, such class is the CollectionLiteralPart.

As we explained when we introduced the concept of collections, we can define collection items by means of a range or a list of items. First way of defining collection items is represented by a CollectionRange, which is a subclass of CollectionLiteralPart.

CollectionRange contains two OclExpressions representing the first and last delimiters of such range. The type of these delimiters should be Integer.

Finally, the other way to represent an item inside a collection is by means of using the CollectionItem metamodel class, which also is subclass of CollectionLiteralPart. Such metamodel class contains a reference to an OclExpression, which represents the expression item of the collection.

To summarize, along this chapter we have seen the different metamodel classes (also known as metaclasses) that conform the OCL 2.0 metamodel used into the Eina GMC environment. Therefore, this will be the metamodel used in our OCL 2.0 expressions processor.

COMPILER BASICS 8

8 COMPILER BASICS

8.1 INTRODUCTION

Since the purpose of this project is to construct an OCL parser, to understand some concepts about compilers is mandatory. In this chapter we introduce concepts like languages, grammars and automatons, as a part of the formal computation theory[CM03], with a few examples in order to make more understandable the next chapters. Furthermore, to know the different phases of the compilation process is important in this context therefore it will be exposed showing the relationships with the development of the OCL parser.

8.2 FORMAL LANGUAGE THEORY

The formal language theory [HMU01] provides the basis of the compiler theory since compilers have language expressions as their input or output. In this section we explain the existing way to cover from words to automatons, that is, from the minimum unity which could be input of a compiler to the mechanisms that a compiler uses to process.

8.2.1 WORDS AND LANGUAGES

Human beings use words every day. Each word, denoted as ω, is a finite sequence of symbols over an alphabet (e.g. the ASCII alphabet of characters or the decimal alphabet whose symbols are the numbers from 0 to 9). It can be said that we are compilers of information given in words and our output is a mixture of feelings, more information to exchange or store, and orders for our nervous system to do different tasks. In other words, we process input data and turn it to another format that we are able to use. This is how a compiler works.

Word(ω) : finite sequence of symbols over and alphabet Σ.

Definition 8.1: Word

Binary alphabet ≡ $\Sigma_1=\{0,1\}$ $\omega_1,\omega_2 \in \Sigma_1$

$\omega_1=01110010$ **and** $\omega_2=110001111$

ASCII alphabet ≡ $\Sigma_2=\{a..z, A..Z, 0..9\}$ $\omega_3,\omega_4 \in \Sigma_2$

ω_3=*Compilation* **and** ω_4=*Formula1*

Example 8.1: Alphabets and words

Similarly, we define languages (L) as any set (finite or not) composed with words ω over a determined alphabet Σ. Following the previous example, human beings work with languages and know the (greater part of) words that conform them. We can see the compilation process as the communication process in a conversation. If we want to achieve a correct communication between two persons, they have to know the same language and use it properly. At the compilation process, if we want a perfect result of the compilation, the compiler has to master the input and output languages.

Language(L): any set (finite or not) composed with words over a determined alphabet (e.g. the English, Spanish or Catalan languages).

Definition 8.2: Language

$\Sigma_1=\{0..9\}$ ***decimal alphabet*** L_1 ***and*** L_2 ***languages over*** Σ_1

$L_1=\{0, 2, 4, 6, 8, 10, 12, \ldots\}$ ≡ ***even numbers***

$L_2=\{1, 3, 5, 7, 9, 11, 13, \ldots\}$ ≡ ***odd numbers***

Example 8.2: Alphabet and languages over it

8.2.2 Operations on languages

In order to work with languages and words there are some operations that allow specifying new ones with more complexity by means of basic constructions or iteratively. Next, we explain the three most common operations that can be found.

8.2.2.1 Union

The union operation ($\cup$) is the common binary operation in set theory that results in the elements that belong to any of the two set sources.

$$L_1 \cup L_2 = \{\omega \mid \omega \text{ is in } L_1 \text{ or } \omega \text{ is in } L_2\}$$

Definition 8.3: Formal definition of union

$\Sigma=\{a, b\}$ $L_1=\{a, ab, bbb, aa\}$ and $L_2=\{aaa, bbaa\}$

$$L_1 \cup L_2 = \{a, ab, bbb, aa, aaa, bbaa\}$$

Example 8.3: Union of two languages

8.2.2.2 Concatenation

In words case, the result of a concatenation of two words (or symbols) is the word composed by the first word followed by the second one without any space between them. The common notation for the concatenation is to use the dot ("·") symbol between the two members of the operation.

$\Sigma=\{a, b, c\}$ *if* $\omega_1=aba$ *and* $\omega_2=cba$ *then* $\omega_1 \cdot \omega_2=abacba$

Example 8.4: Concatenation of two words

On the other hand, in languages case, the result of a concatenation of two languages is the language that contains words composed by a word from the first language concatenated with a word from the second language.

$$\Sigma=\{a, b\} \text{ if } L_1=\{a, bb, aba\} \text{ and } L_2=\{a, cc\}$$
$$\text{then } L_1{\cdot}L_2=\{aa, acc, bba, bbcc, abaa, abacc\}$$

Example 8.5: Concatenation of two languages

8.2.2.3 Kleene star

Also known as Kleene closure, this operation is related with the concatenation, which was defined previously. The Kleene star of a language or word results in the set containing the repeated concatenation of them with the same language or word respectively. In order to be more explicit, we introduce the following notation:

L is a language, ω is a word and λ is the empty word, which length is zero.

$$L^0=\{\lambda\} \text{ and } \forall i\geq 0 \;\; L^{i+1} = L^i \cdot L$$
$$\omega^0=\lambda \;\text{ and } \forall i\geq 0 \;\; \omega^{i+1} = \omega^{\,i} \cdot \omega$$
$$\text{Then, } L^*=L^0\cup L^1\cup L^2\cup L^3\cup L^4\cup L^5\cup\cdots\cup L^n$$
$$\text{and } \omega^*=\omega^0\cup\omega^1\cup\omega^2\cup\omega^3\cup\omega^4\cup\omega^5\cup\cdots\cup\omega^n$$

Definition 8.4: Exponential notation of repeated concatenation and Kleene star definition

These expressions allows us to show the result of the concatenation of languages and words with themselves i times, using an exponential notation. Therefore, the Kleene star must be represented as L^* and ω^*, indicating that the number of repeated concatenations is undefined. Applying Kleene star to a finite language or a word results in an infinite one because of the infinity of the number of concatenations.

$\Sigma=\{a, b\}$ if $L=\{a, bb, aba\}$ then $L^*=\{\lambda, a, aa, bb, aaa, abb, bba, \dots\}$

where $L^*= L^0 \cup L^1 \cup L^2 \cup L^3 \cup \cdots \cup L^n$

and $L^0= \{\lambda\}$, $L^1= \{a, bb, aba\}$, $L^2= \{aa, abb, aaba, bba, bbbb, bbaba, abaa, ababb, abaaba\}$ and so on …

Example 8.6: Kleene star operation applied to a three-word language.

8.2.2.4 Kleene plus

Similarly to the Kleene star, the Kleene plus, also known as positive Kleene closure, is defined as follows:

$$L^+ = L^* - \{\lambda\}$$

$$\omega^+ = \text{any } \omega^i \text{ where } i \geq 1$$

Definition 8.5: Formal notation of Kleene plus operation

It indicates that L^+ results in the same set as the Kleene star but without the empty word and ω^+ results in a word that must have length greater than zero, i.e., it cannot be the empty word.

$\Sigma=\{a, b\}$ if $L=\{a\}$ then $L^+=\{ a, aa, aaa, aaaa, aaaaa, \dots\}$

if $L=\{a, bb, aba\}$ as in Example 8.6

then $L^+=\{ a, aa, bb, aaa, abb, bba, \dots\}= L^* - \{\lambda\}$

Example 8.7: Kleene plus applied to both single-word and three-word languages

8.2.3 Regular expressions

Regular expressions are expressions that represent (part of) a language and define it by means of the combined use of operations, words and languages. Here we present the rules that define a regular expression over an alphabet Σ:

1. λ is a regular expression.

2. A symbol s in alphabet Σ is a regular expression.

3. If r_1 and r_2 are regular expressions, then $r_1 \cup r_2$ and $r_1 \cdot r_2$ are regular expressions.

4. If r is a regular expression, then r^* and r^+ are regular expressions.

5. All regular expressions over Σ are made applying rules 1 to 4.

Definition 8.6: Rules to construct regular expressions

We define *L(r)*, as associated language *L* to a regular expression *r*, to the language recognized by *r*.

$r_1=a^*$ then $L(r_1)=\{\lambda, a, aa, aaa, aaaa, aaaaa, \ldots\}$

$r_2=ba^*b$ then $L(r_2)=\{bb, bab, baab, baaab, baaaab, baaaaab, \ldots\}$

$r_3=(ba)^+a^*$ then $L(r_3)=\{ba, baa, baaa, baba, baaaa, babaa, \ldots\}$

$r_4=b^* \cup a^*$ then $L(r_4)=\{\lambda, b, a, bb, aa, bbb, aaa, bbbb, aaaa, \ldots\}$

Example 8.8: Regular expressions and their associated languages

8.2.4 GRAMMARS AND AUTOMATONS

Another way to recognize and represent a language is by means of grammars and automatons. They are formal mechanisms used in different phases of the compilation process, as we will see in next chapters. To write a grammar is one of the most common ways to begin the construction of a compiler because of the existence of compiler compilers. These tools have a grammar as input and generate the full compiler (semi) automatically. Every word that can be constructed through the grammar belongs to the recognized language of it.

A simple grammar *G* is a formal structure like $G=\langle V, \Sigma, P, S \rangle$ where their components are:

· V is an alphabet which symbols are called variables.

· Σ is another alphabet, disjoint of V, which symbols are called terminals.

· P is a set of production rules $A \Rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$

· S is called start variable.

Definition 8.7: Grammar components

Grammar G=⟨ V, Σ, P, S ⟩

V={S, T, X, Y, Z} are the variables

Σ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, x, (,)} are the terminals

The productions P are:

S→ X | X + S

X→ T | Y x Z

Y→ T | (X + S)

Z→ Y | Y x Z

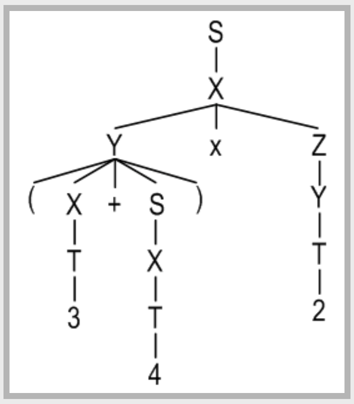
T→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

With this grammar we can recognize the language of aritmetical expressions with add or product operations. The | caracter separates different alternatives for the right side of a production rule.

For example, the expression "(3+4) x 2" can be derivated as follow:

S ⇒ X ⇒ Y x Z ⇒ (X + S) x Z ⇒ (T + X) x Z ⇒ (3 + T) x Z ⇒ (3 + 4) x Z ⇒ (3+4) x T ⇒ (3+4) x 2

Or in tree form:

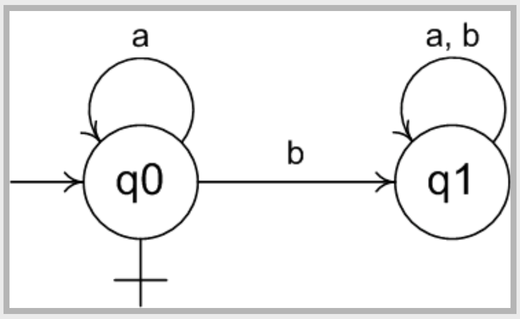


Example 8.9: Grammar and recognized language

In the context of this document, when we use automaton as word we are talking about finite automatons. They can be deterministic or non-deterministic depending on whether their state traversal is a single path or multiple one, that is, if starting in a state we can only move to another one or to a set of them.

A deterministic finite automaton (DFA) is a formal structure drawn like a graph where their states are crossed according to the symbols of an input word. There are one start state and one or more final states. If the input word crossing ends in a final state, the automaton recognizes the word, i.e., it belongs to the associated language.

Graph representing the automaton:



There are two states q0 and q1, represented with a circle. The start state is q0, denoted with the first left arrow. The arcs from one state to another one indicate the way to follow for every symbol of the word in processing.

For example, for the word "aba", we begin with the first symbol 'a' at the state q0. Then, we go to the state q1 with the second symbol 'b'. Finally, the last symbol is 'a', so we continue at state q1.

Every state with a cross is a final state. If when we finish the symbols of the input word we are in a final state, the word is recognized by the automaton and belongs to the language.

The automaton of the image recognizes words that belong to the language a*. It is easy to understand because when a 'b' arrives we go to a non-final state, which we cannot leave. This language is not a^+ because the start state is already a final state, so even the empty word is accepted.

Example 8.10: DFA recognizing the language a*

8.3 THE COMPILATION PROCESS

Once we have seen and understoodd the basic concepts, we are ready to begin with the explanation of how the compilation process[WM95] works. In the image below the different phases of the process as well as the connexions between them can be distinguished.

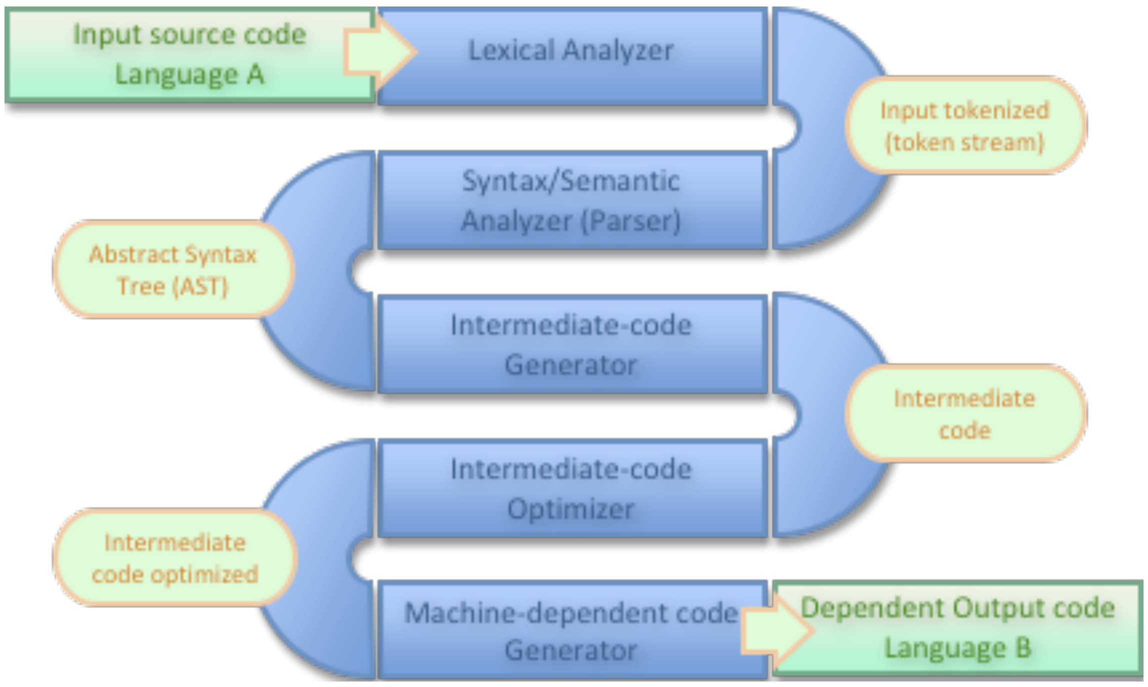

Input source code Language A
Lexical Analyzer
Input tokenized (token stream)
Syntax/Semantic Analyzer (Parser)
Abstract Syntax Tree (AST)
Intermediate-code Generator
Intermediate code
Intermediate-code Optimizer
Intermediate code optimized
Machine-dependent code Generator
Dependent Output code Language B

Figure 8.1: Compiler phases

8.3.1 LEXICAL ANALYSIS

The main purpose of this phase is to partition and classify the input of the compiler (see Example 8.11). It is made by means of tokens.

A token is a lexical component that has a coherent meaning in some language. They are the most basic elements on which all translation of a program is based. Examples of tokens are the key words (e.g., if, while or int), identifiers, numbers, signs or operators, in the context of programming languages.

Token: identifies a unit of information.

Usually, "tokens" are the result of some processing pass that has performed lexical analysis and divided a data set into the smallest units of information used for subsequent processing.

Definition 8.8: Token

Input: `context Person inv mustBeAdult: age >= 18`

Tokens:

```
CONTEXT   :    context
ID        :    (a..zA..Z)+
INV       :    inv
COLON     :    ":"
GREQTHAN  :    ">="
NUMBER    :    (0..9)+
```

Output: `CONTEXT ID INV ID COLON ID GREQTHAN NUMBER`

Example 8.11: Lexical analysis with tokens (input, tokens and output tokenized)

8.3.2 Syntax analysis

In this phase, also known as parsing, the input is the main input code tokenized at the previous phase and the output is a parse tree, whose nodes represent the input with more additional information that allows an easiest scanning in order to work with it and to make the correct translation process. Commonly, the parse tree is known as concrete or abstract syntax tree (CST or AST) and it is created thanks to a grammar specification.

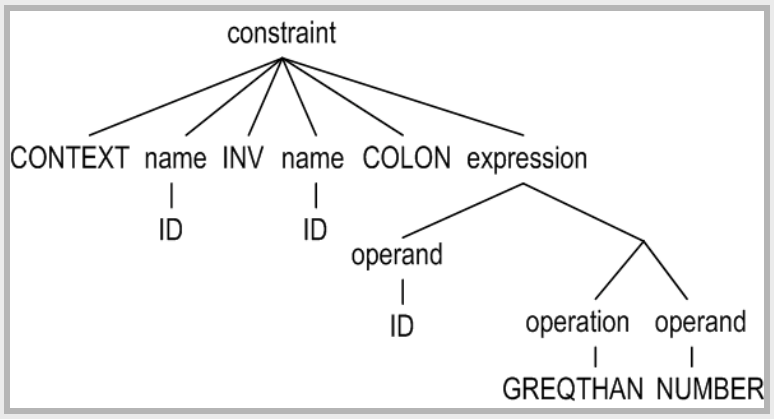
The compiler analyzes the tokens of the input and the different productions of the grammar in order to construct the corresponding parse tree following those rules. A little example can be found at Example 8.12.

Input: `CONTEXT ID INV ID COLON ID GREQTHAN NUMBER`

Grammar productions:

```
constraint   = CONTEXT name INV name COLON expression
name         = ID
expression   = operand (operation operand)*
operation    = GREQTHAN
operand      = ID | NUMBER
```

Parse Tree:



Example 8.12: Syntactic analysis (input tokenized, grammar productions and output parse tree)

8.3.2.1 CST or AST

Commonly, the parse tree can be named as Concrete Syntax Tree (CST) or Abstract Syntax Tree (AST). It depends on the process used to develop the compiler. Usually, we obtain a CST, that is, our parse tree, and then we transform it to an AST.

An AST differs from a CST by omitting nodes and edges for syntax rules of the grammar that do not affect the semantics of the output language. Only significant output language constructs are included. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is implicit in the tree structure. The CST usually is the output of syntax analysis and AST the one of semantic analysis.

For example, the parse tree at Example 8.12 is a CST because has information about irrelevant tokens like CONTEXT or COLON. The corresponding AST should only have the constraint name, the constrained element name and the expression.

8.3.3 SEMANTIC ANALYSIS

This phase tries to obtain that the parse tree does not have errors in order to continue with the translation process of the compilation. If any error is found, the process is stopped and a notification is thrown to the user of the compiler with the corresponding feedback that can help to solve it.

To search every possible error can be a huge task, but is necessary to help the user in his work with the compiler. The more the compiler offers feedback to users, the more usable become it to them.

To achieve a minimum degree of usability, an actual compiler should show for every error found, a simple description and the place where the error happens. Sometimes, a short advice to solve it is always welcome.

8.3.4 CODE GENERATION

Once we have a parse tree decorated with some extra information and empty of errors, the code generation phase is ready to begin. At this point, for every kind of node in the parse tree, a translation to the output language has to be defined in order to convert the input. For instance, when we compile some programming code, a translation is being done from the programming language to the machine code that the computer can interpret and execute.

To know all the possible translations and formats from one language to the output one, in order to obtain the best results, is mandatory.

8.4 COMPILER ARCHITECTURE

In the figure below we can distinguish a complete compiler structure[WM95] with the different phases of work.

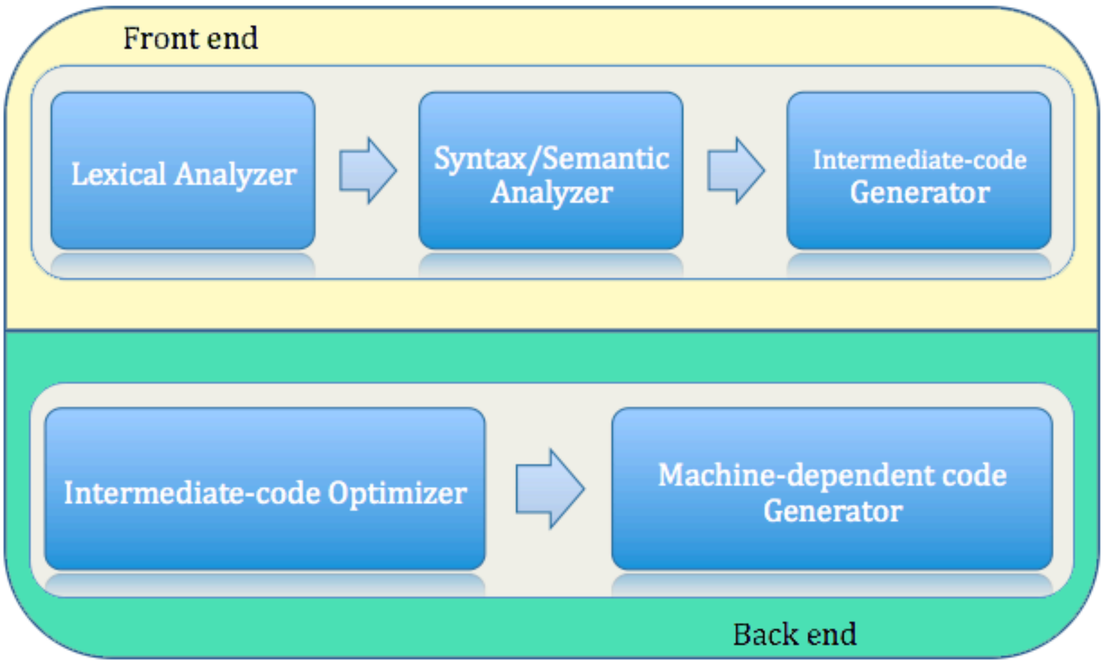

Front end
Lexical Analyzer
Syntax/Semantic Analyzer
Intermediate-code Generator
Intermediate-code Optimizer
Machine-dependent code Generator
Back end

Figure 8.2: Front-end and back-end of a compiler

Our purpose is to develop a front-end compiler that translate OCL2.0 code into XMI according to the OCL2.0 metamodel shown on previous chapters. We need neither a code optimizer nor a machine code translation because our output will not be executed.

Our XMI output might be considered as an intermediate code that could be used by several tools. To share models information is one of our purposes.

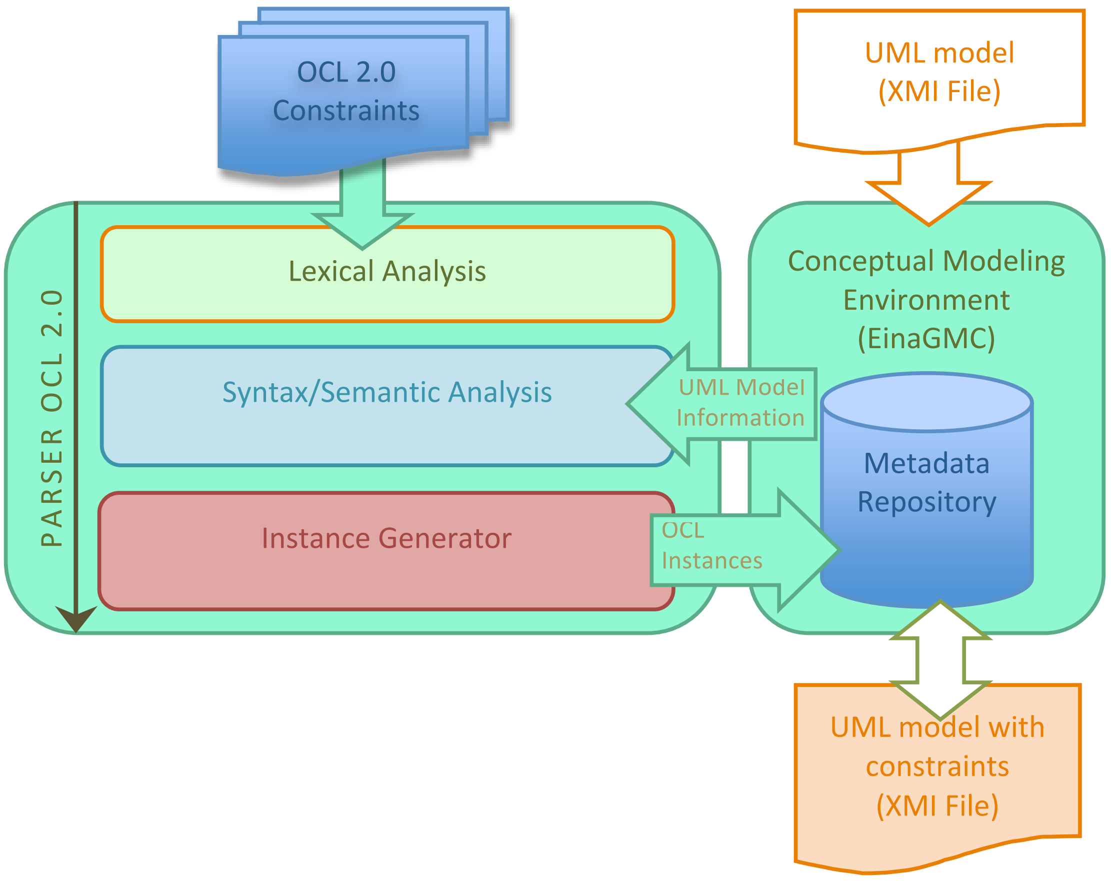


Figure 8.3: Parser OCL 2.0 Architecture

Following this idea, our desire is to develop the front-end of an OCL2.0 compiler (sometimes we will refer to it as parser) which output should be in a standard language format (we have chosen XMI according to MOF). An outline of the architecture can be seen in the following image.

At Figure 8.3 there are two main structures to analyze. Firstly, the conceptual modeling environment to use, named Eina GMC, in which we emphasize the metadata repository, containing both UML and OCL 2.0 metamodels. In addition, it also contains all the instances over them representing the input UML model. All the classes, attributes and so on UML models are instantiated as objects in the repository. This information is necessary for the syntax and semantic analysis of our intended OCL parser in order to verify the correctness of the constraints.

Secondly, our OCL Parser itself has the same phases as we have seen before in the chapter. However, the intermediate-code generator phase is called instance generator phase here because of our intermediate language are the OCL2.0 instances of the metadata repository.

Finally, we have to remember that with a UML model instances information and a OCL 2.0 set of constraints we will be able to verify the correctness of the constraints and then generate their instantiation on the repository, in order to obtain a UML model with constraints in a standard language (XMI) that can be shared between the users, thanks to our OCL 2.0 parser/compiler.

OCL TOOLS 9

9 OCL TOOLS

9.1 STATE OF THE ART

There exist some tools that are capable of supporting and handling OCL expressions. These tools aim at making this language easier to use for analysts and developers who already use the language, as well as to those who intend to use it in the near future.

Along this chapter we will introduce some of these tools in order to choose the OCL tool which best adapts to our requirements. It is important to remember that our aim is to develop a processor of OCL 2.0 expressions in order to convert textual constraints into metamodel instances. Therefore, to choose a tool that seems to be similar than our processor in mind will help us to begin the development of it with a good basis.

Nowadays to start a software development process from scratch is not a good idea. As in our case, it is very easy to find people with our needs that thought solutions to our problems before us, and probably own a software tool that could be similar than our desired tool. Therefore, to do a little effort searching similar solutions to our problem could help us decreasing the development time and the complexity of our project.

Papers, articles and the whole Internet are good places where to do our searching task. If exists a (partial) solution to our problem, probably it has a web page where its functionalities and other features are described. So, in the next section we will study some tools that work with OCL expressions and could be useful to our development phase.

9.2 OCL TOOLS

After searching on the Internet, we have chosen a set of tools that work with OCL expressions and could be (partially) similar to our processor.

At next subsections we will discover their functionalities and features that are provided. Moreover, a link to the place where to find each alternative is given in order to allow readers of this document to test these tools by themselves.

9.2.1 UML-BASED SPECIFICATION ENVIRONMENT (USE)

As indicated in [USEw] USE is a system for the specification of information systems developed inside the Database and Systems Group of the Bremen University. It is based on a subset of the Unified Modeling Language (UML).

A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements.

Such USE specification files have a special syntax. USE does not support import or export models from/to XMI format. Its syntax for declare a model is shown in Example 9.1

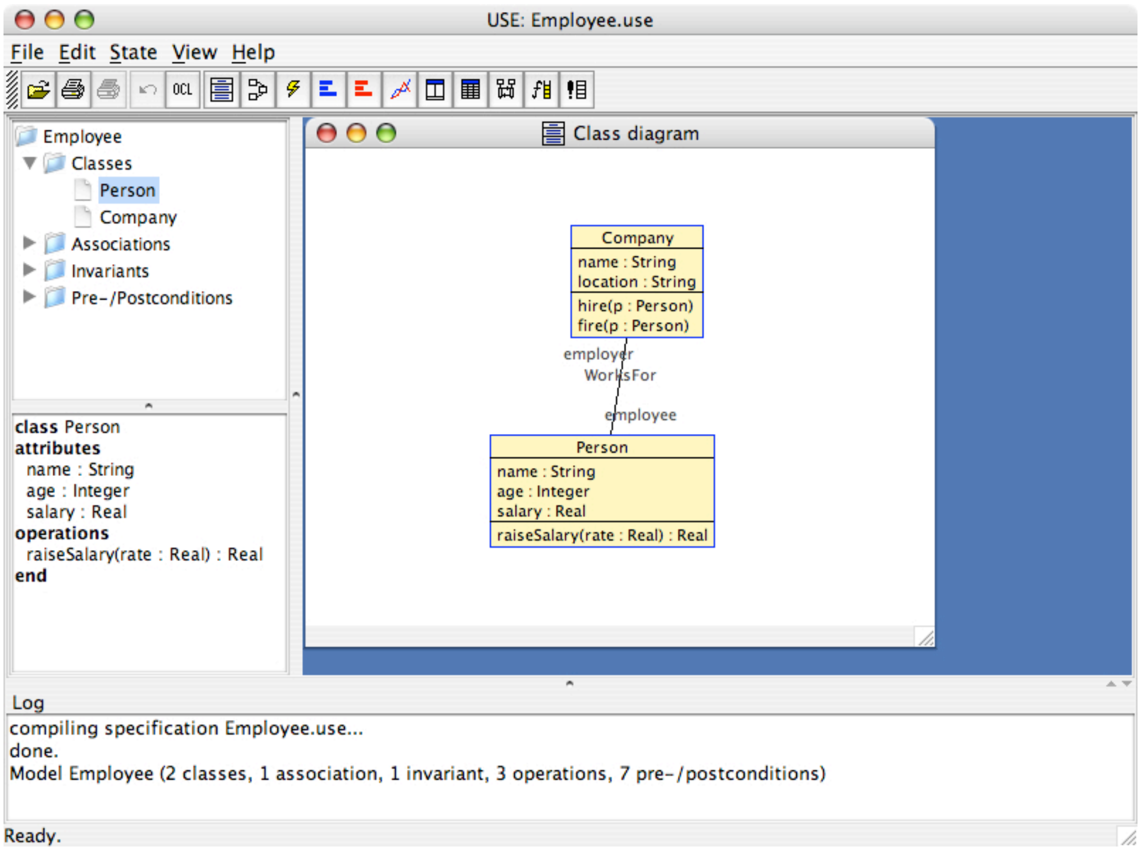
```
model Employee
-- classes
class Person
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end
class Company
attributes
  name : String
  location : String
operations
  hire(p : Person)
  fire(p : Person)
end
-- associations
association WorksFor between
  Person[*] role employee
  Company[0..1] role employer
end
```

Example 9.1: USE specification file extracted from [USEw]

Keywords are emphasized with bold notation in order to note the different sections of the specification syntax.

System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state.

USE allows adding real instances to UML classes in order to do such evaluation of the system and their constraints. These instance objects represents a system state that is evaluated in order to check its correctness.



```
class Person
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end
```

```
compiling specification Employee.use...
done.
Model Employee (2 classes, 1 association, 1 invariant, 3 operations, 7 pre-/postconditions)
```

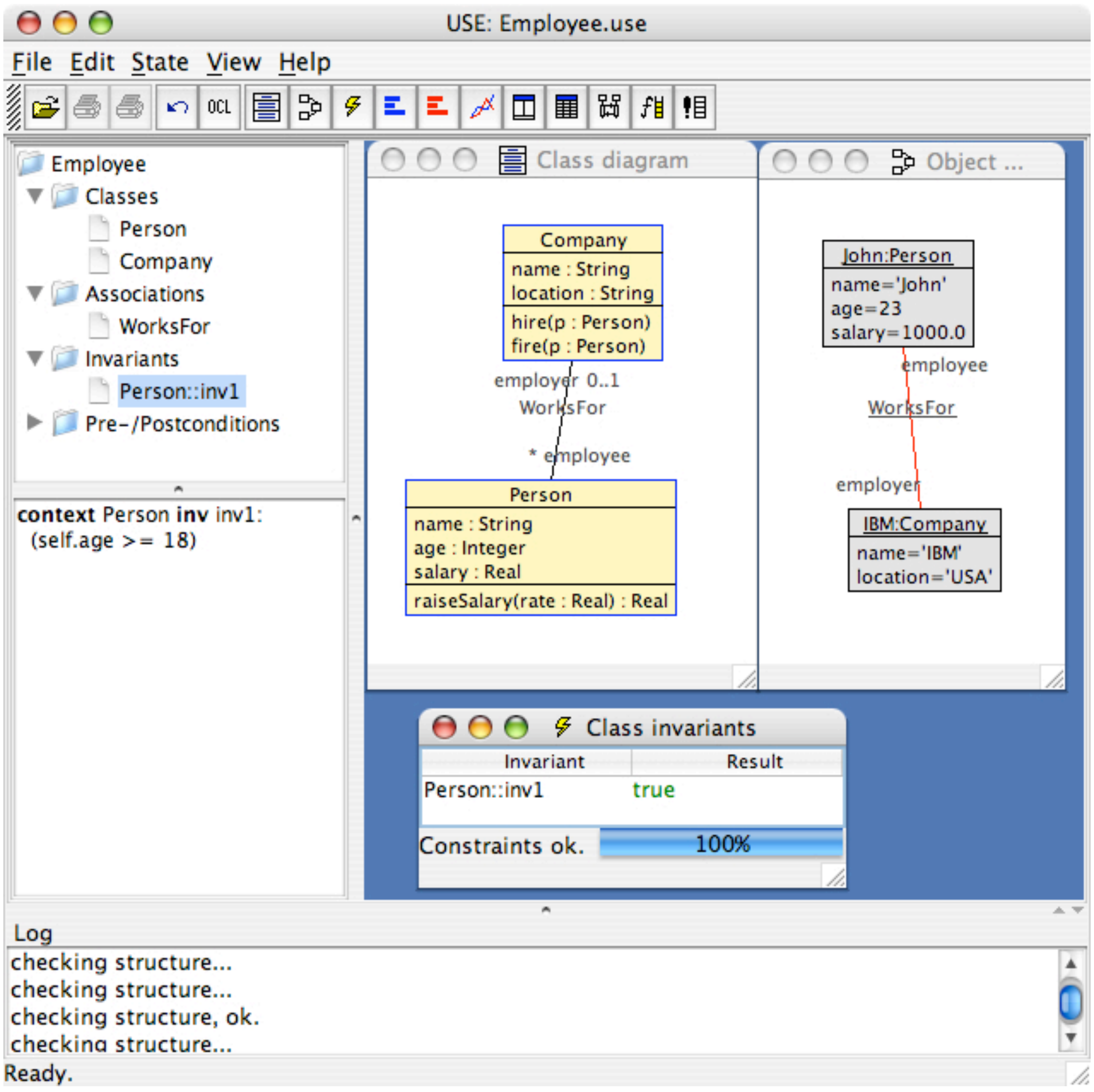
Figure 9.1: USE graphical user interface

USE can be executed by a command line interface (CLI) or through a graphical user interface (GUI). Anyways the first step is to load a USE specification file with the model description. Such file can also contain OCL constraints into another section after the previous one shown at Example 9.1 like in next example.

```
-- OCL constraints
constraints
context Person inv inv1: self.age >= 18
```

Example 9.2: Constraints section of a USE specification file

Once we have loaded the specification file in USE format we will obtain a window like in Figure 9.1. It contains a view of packages with the elements of the model and USE also provides different views as class or object diagrams. At Figure 9.2 we can find the USE interface with three windows including a class diagram with the two classes indicated into the specification file, an object window with the instances of the previous classes and relationships, and finally a invariant with the result of applying the invariant constraints to the object instances.



```
context Person inv inv1:
  (self.age >= 18)
```

Invariant	Result
Person::inv1	true

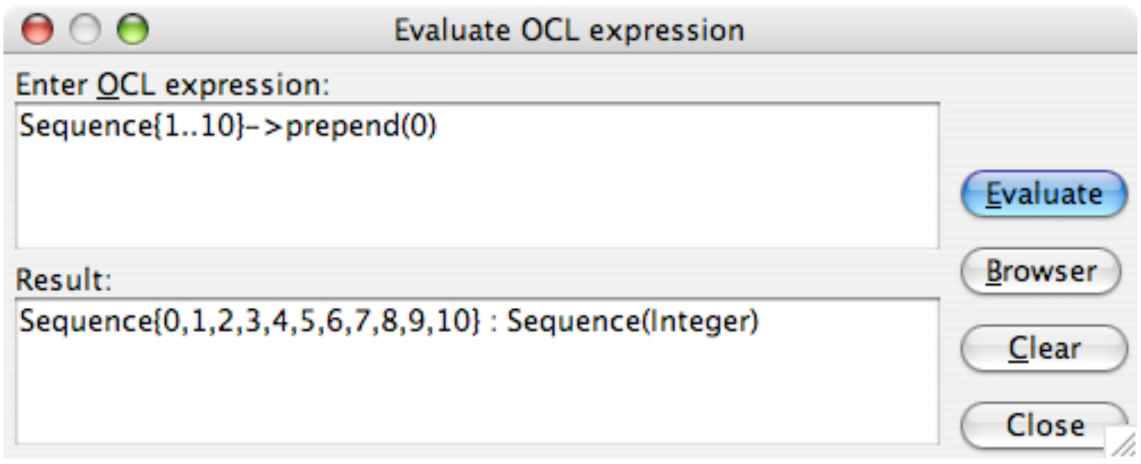
Constraints ok. 100%

```
checking structure...
checking structure...
checking structure, ok.
checking structure...
```

Figure 9.2: Different views inside the USE interface

Inside the USE window it is possible to create object instances but not classes or relationships to add to the model specified before.

The OCL parser and interpreter of USE allows the evaluation of arbitrary OCL expressions. The menu item *Evaluate OCL expression* opens a dialog where expressions can be entered and evaluated. An example is shown at next image. The direct interpretation of OCL expressions is the best feature of this tool.



```
Sequence{1..10}->prepend(0)
```

```
Sequence{0,1,2,3,4,5,6,7,8,9,10} : Sequence(Integer)
```

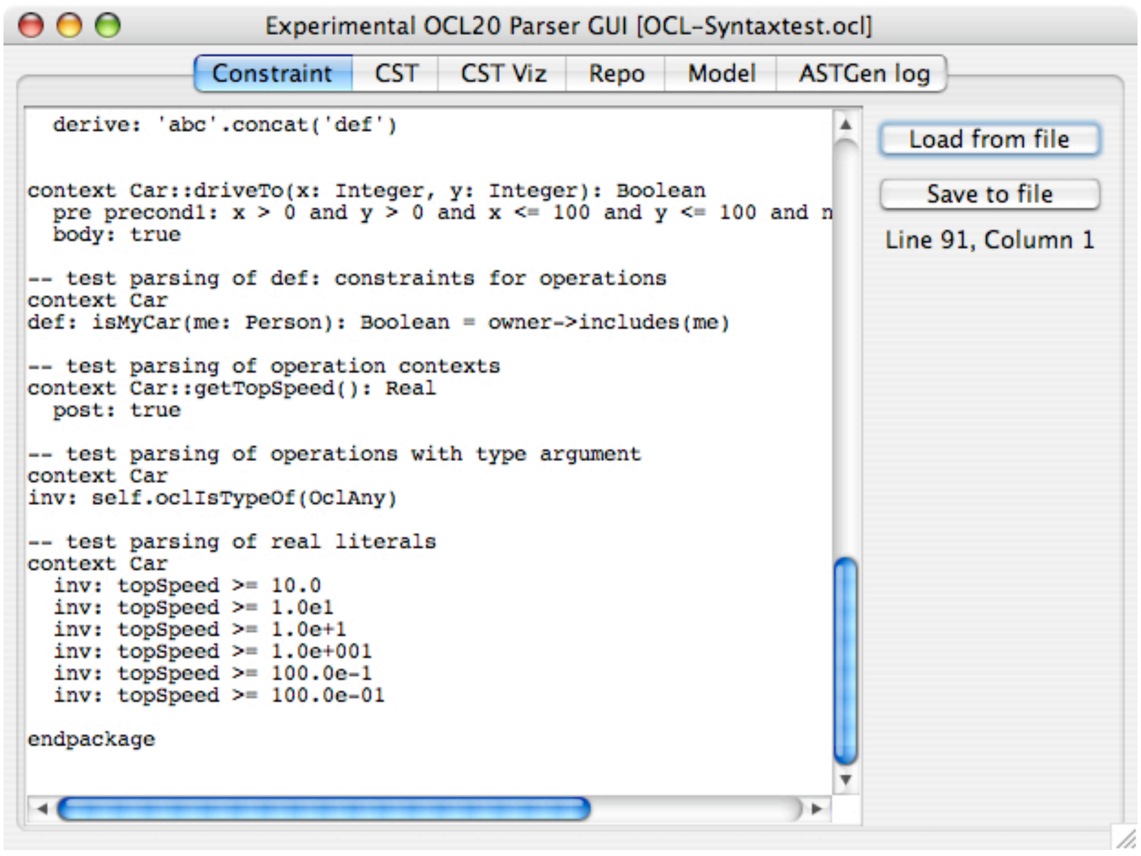
Figure 9.3: OCL expression checking inside USE

Since USE contains a subset of the complete UML and OCL languages, some constructions or operations are not included in its implementation. For example, *OrderedSet* is not a known kind of Collection or operations like *insertAt* are not recognised.

USE is implemented in Java and its distribution comes with full sources. A final version (last is 2.4.0) can be downloaded from [USEw]. Furthermore, a whole explanation about how to use all USE functionalities can be found at its user guide in [USE07].

9.2.2 Dresden OCL2 Toolkit

The Dresden OCL2 Toolkit is a complete framework composed by several components. In this section we are interested in the OCL 2.0 parser subsystem, which consists of a number of modules that interoperate to create an abstract syntax tree from textual OCL 2.0 constraints. It is currently developed by members of the Technische Universität Dresden. The OCL 2.0 parser subsystem was developed by Ansgar Konermann [DOTw].

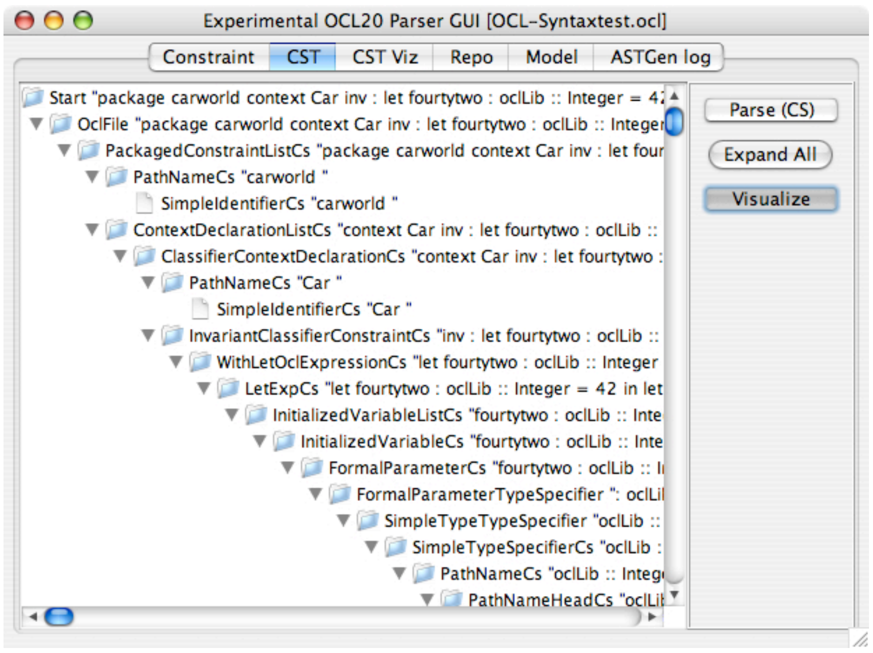


```
  derive: 'abc'.concat('def')


context Car::driveTo(x: Integer, y: Integer): Boolean
  pre precond1: x > 0 and y > 0 and x <= 100 and y <= 100 and n
  body: true

-- test parsing of def: constraints for operations
context Car
def: isMyCar(me: Person): Boolean = owner->includes(me)

-- test parsing of operation contexts
context Car::getTopSpeed(): Real
  post: true

-- test parsing of operations with type argument
context Car
inv: self.oclIsTypeOf(OclAny)

-- test parsing of real literals
context Car
  inv: topSpeed >= 10.0
  inv: topSpeed >= 1.0e1
  inv: topSpeed >= 1.0e+1
  inv: topSpeed >= 1.0e+001
  inv: topSpeed >= 100.0e-1
  inv: topSpeed >= 100.0e-01

endpackage
```

Figure 9.4: Main window of the OCL 2.0 parser subsystem of the Dresden OCL2 Toolkit

As we can see at Figure 9.4, the graphical interface of such parser is divided into 6 different sections. First one is implemented to write OCL 2.0 constraints or also load/save them from/to a file.

Next step is to parse the OCL 2.0 constraints in order to obtain a concrete syntax tree (CST). This process is similar than such explained at chapter 8 when describing compilation process.



```
Start "package carworld context Car inv : let fourtytwo : oclLib :: Integer = 42
▼ OclFile "package carworld context Car inv : let fourtytwo : oclLib :: Integer
  ▼ PackagedConstraintListCs "package carworld context Car inv : let four
    ▼ PathNameCs "carworld "
        SimpleIdentifierCs "carworld "
    ▼ ContextDeclarationListCs "context Car inv : let fourtytwo : oclLib ::
      ▼ ClassifierContextDeclarationCs "context Car inv : let fourtytwo :
        ▼ PathNameCs "Car "
            SimpleIdentifierCs "Car "
        ▼ InvariantClassifierConstraintCs "inv : let fourtytwo : oclLib ::
          ▼ WithLetOclExpressionCs "let fourtytwo : oclLib :: Integer
            ▼ LetExpCs "let fourtytwo : oclLib :: Integer = 42 in let
              ▼ InitializedVariableListCs "fourtytwo : oclLib :: Inte
                ▼ InitializedVariableCs "fourtytwo : oclLib :: Inte
                  ▼ FormalParameterCs "fourtytwo : oclLib :: I
                    ▼ FormalParameterTypeSpecifier ": oclLi
                      ▼ SimpleTypeTypeSpecifier "oclLib ::
                        ▼ SimpleTypeSpecifierCs "oclLib :
                          ▼ PathNameCs "oclLib :: Integ
                            ▼ PathNameHeadCs "oclLi
```

Figure 9.5: Parsed constraints into CST format

It is important to note that the CST can be shown in a hierarchical package form or in a graphical notation.

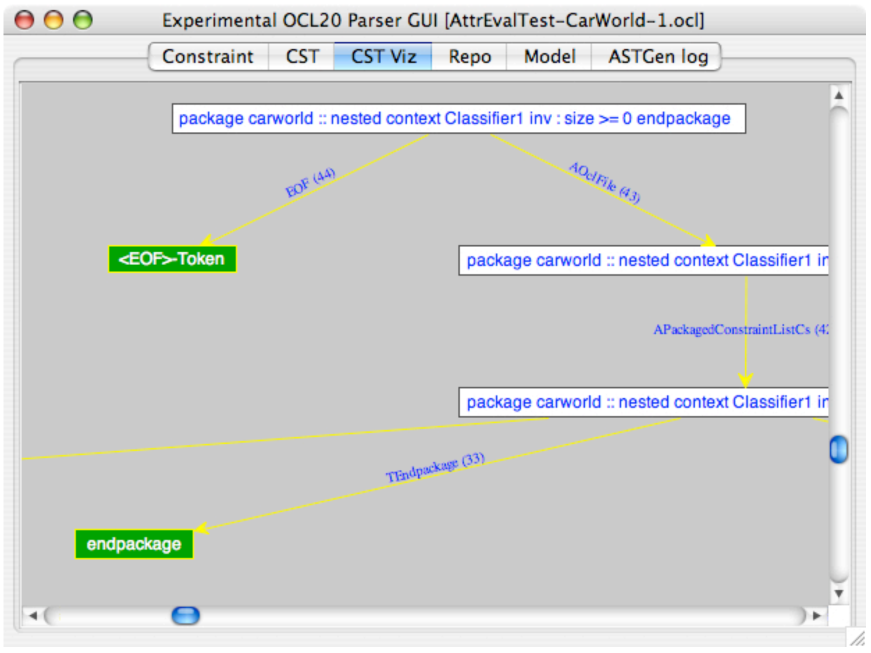
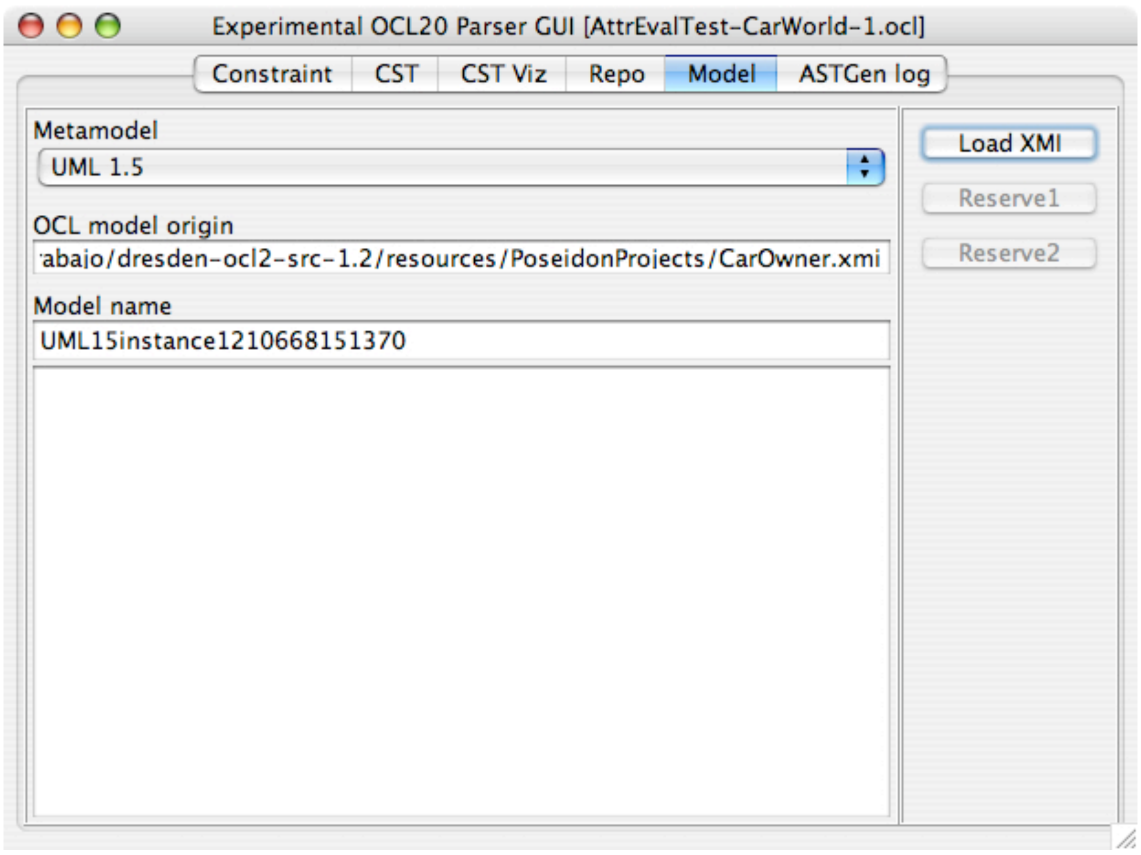

Experimental OCL20 Parser GUI [AttrEvalTest-CarWorld-1.ocl]
Constraint
CST
CST Viz
Repo
Model
ASTGen log
package carworld :: nested context Classifier1 inv : size >= 0 endpackage
EOF (44)
AOclFile (43)
<EOF>-Token
package carworld :: nested context Classifier1 in
APackagedConstraintListCs (42
package carworld :: nested context Classifier1 in
TEndpackage (33)
endpackage

Figure 9.6: Visualization of CST generated by OCL 2.0 parser subsystem

Once we have the CST, the OCL 2.0 parser subsystem needs to load a UML model to check if the OCL constraints are correctly constructed. Into model section of the GUI we can load an XMI file representing the UML model.

It is important to emphasize that UML version allowed is 1.5 instead of the last 2.0. Nevertheless, the version of OCL supported is 2.0.



Metamodel
UML 1.5

OCL model origin
'abajo/dresden-ocl2-src-1.2/resources/PoseidonProjects/CarOwner.xmi

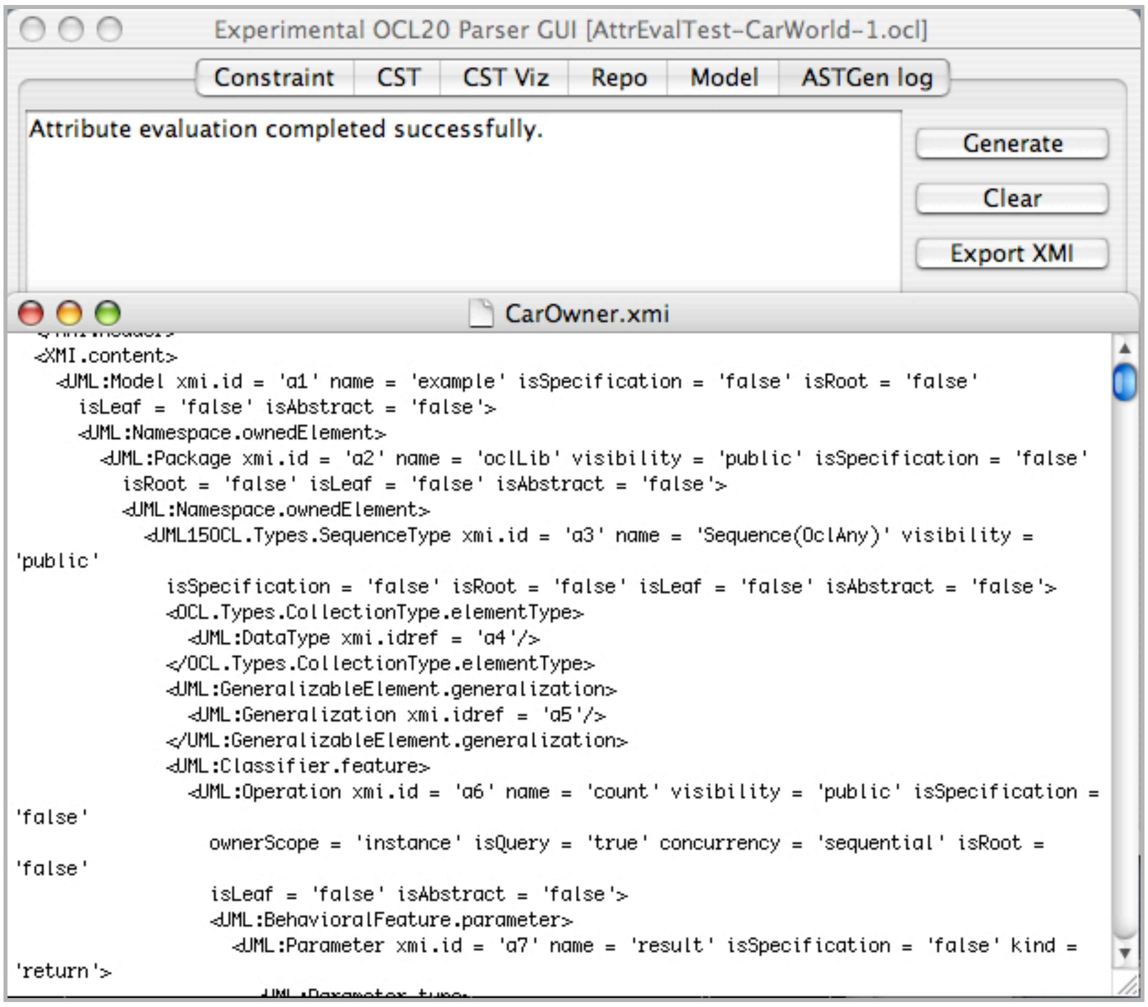
Model name
UML15instance1210668151370

Figure 9.7: XMI model loaded within the OCL 2.0 parser subsystem

Finally, last step in the workflow of the OCL 2.0 parser subsystem is to generate the evaluation of the constraints according to the UML 1.5 model loaded. If any error is found at this process, this tool informs the user in order to solve it.

One of the most important features of the OCL 2.0 parser subsystem is that it provides the possibility of exporting the current UML 1.5 model with the new constraints into an XMI file, according to the standard metamodels.

Such functionality is very similar than our intention of instantiating textual constraints and then adding them into the XMI model file.



Attribute evaluation completed successfully.

```
  <XMI.content>
    <UML:Model xmi.id = 'a1' name = 'example' isSpecification = 'false' isRoot = 'false'
      isLeaf = 'false' isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Package xmi.id = 'a2' name = 'oclLib' visibility = 'public' isSpecification = 'false'
          isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
          <UML:Namespace.ownedElement>
            <UML15OCL.Types.SequenceType xmi.id = 'a3' name = 'Sequence(OclAny)' visibility =
'public'
              isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
              <OCL.Types.CollectionType.elementType>
                <UML:DataType xmi.idref = 'a4'/>
              </OCL.Types.CollectionType.elementType>
              <UML:GeneralizableElement.generalization>
                <UML:Generalization xmi.idref = 'a5'/>
              </UML:GeneralizableElement.generalization>
              <UML:Classifier.feature>
                <UML:Operation xmi.id = 'a6' name = 'count' visibility = 'public' isSpecification =
'false'
                  ownerScope = 'instance' isQuery = 'true' concurrency = 'sequential' isRoot =
'false'
                  isLeaf = 'false' isAbstract = 'false'>
                  <UML:BehavioralFeature.parameter>
                    <UML:Parameter xmi.id = 'a7' name = 'result' isSpecification = 'false' kind =
'return'>
```

Figure 9.8: Attribute evaluation and XMI generation within the OCL 2.0 parser subsystem

To conclude, it is important to know that the Dresden OCL2 Toolkit contains other tools that work with OCL 2.0 expressions. An example of such tools is the OCL

Editor, which provides a text editor for OCL constraints, or the OCL22Java that generates Java code from OCL expressions.

9.2.3 MOVA PROJECT

The MOVA Project is a software development project developed by the MOVA (Modeling and Validation) Group that has the MOVA tool as a result.

According to the explanation extracted from their web site [MOVw], the MOVA tool is a modeling and validation experimental tool developed at the Universidad Complutense de Madrid by the MOVA group. The MOVA tool is part of a broader effort for integrating rigorous modeling and validation into the industrial software engineering process.

The MOVA tool consists of three applications:

- UML modeling: it allows the user to draw UML class and object diagrams, write and check OCL invariants, write and evaluate OCL queries, and define OCL operations to be used in invariants and queries.

- SecureUML modeling: it allows the user to draw SecureUML diagrams, write and evaluate OCL security policies, and define OCL operations to be used in security policies.

- UML modeling with metrics: it allows the user draw UML class and object diagrams, write and check OCL invariants, write and evaluate OCL queries, write and evaluate OCL metrics, and define OCL operations to be used in invariants, queries, and metrics.

We are interested only in the first application included in the MOVA tool related to UML modeling. First step when executing MOVA tool is to choose the UML modeling option in order to start such application.

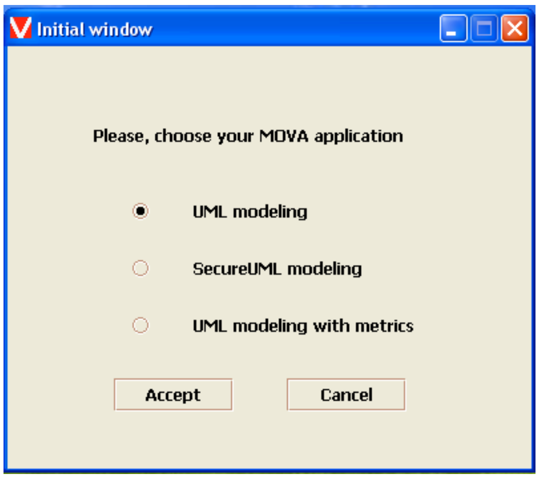

Initial window
Please, choose your MOVA application
● UML modeling
○ SecureUML modeling
○ UML modeling with metrics
Accept
Cancel

Figure 9.9: MOVA tool initial window

Once we have opened the correct application, next step is to load or draw a UML class diagram. MOVA uses its own XML format to load and save models, which is very simple and different from XMI.

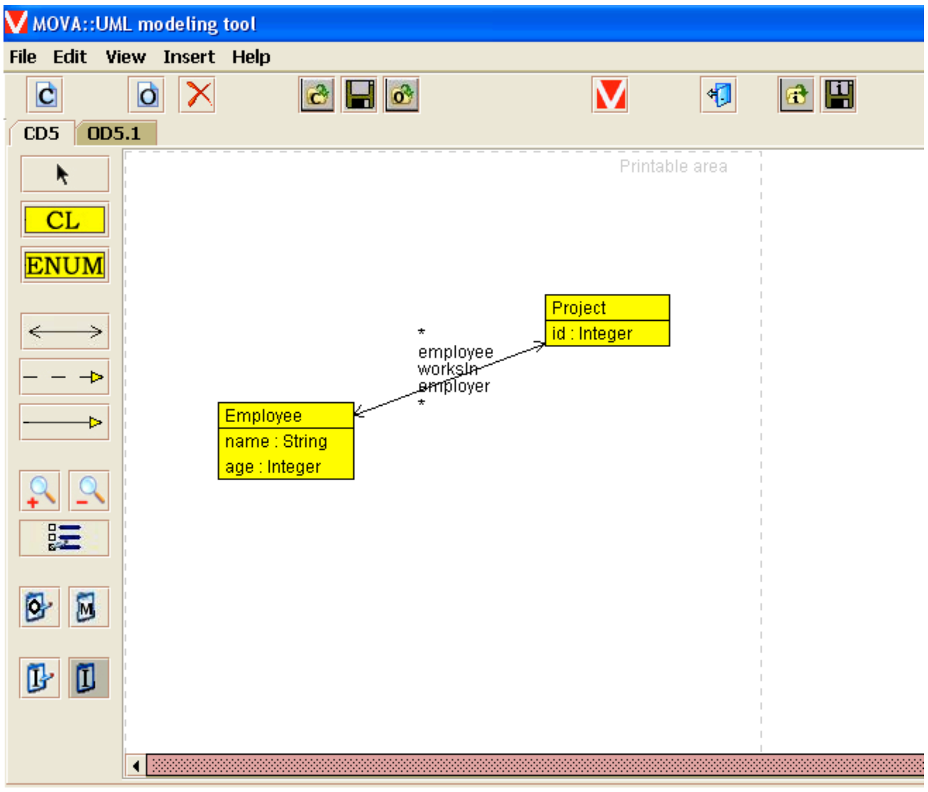


Figure 9.10: MOVA UML modeling tool. Class diagram

MOVA provides an area to draw classes, relations and hierarchies. Furthermore it is possible to create object instances that will be useful to evaluate OCL constraints over them.

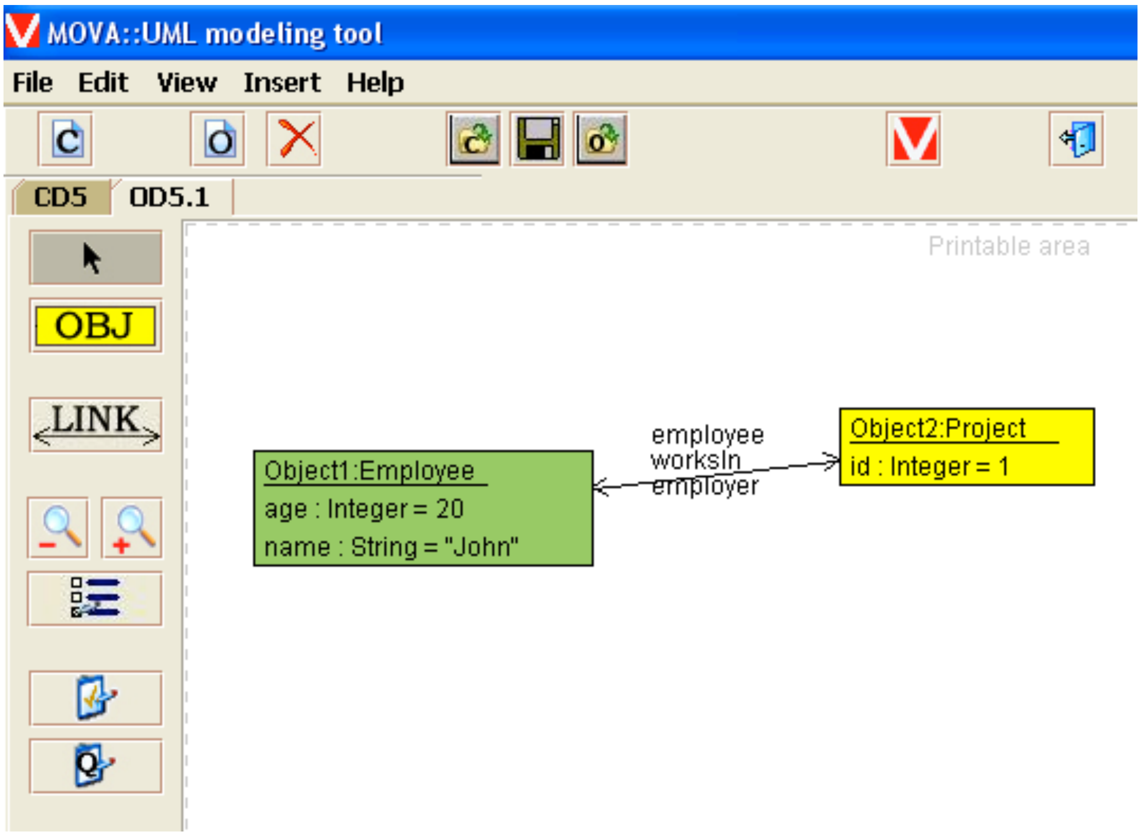
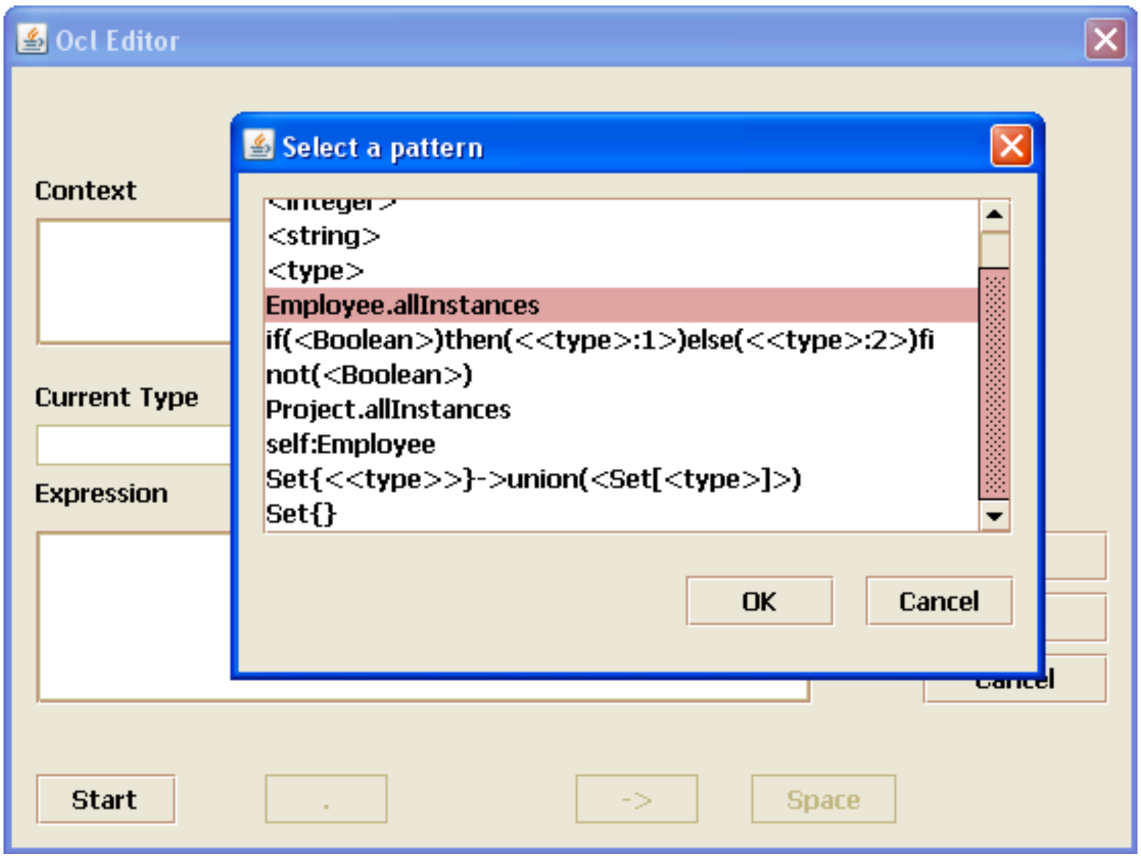


Figure 9.11: Object instantiation inside MOVA tool

The process to create instances is as simple as draw classes. We only have to select which class will be the container of the instance and then complete the attributes and relationships of such instance. An example of instantiation can be found at Figure 9.11

When we have defined both class and object diagrams it is the moment to write the OCL constraints over the class elements. MOVA tool provides a step-by-step window where to write constraints. We must select our current start point of a constraint and then MOVA shows us what are the possibilities to choose in order to complete the expression.

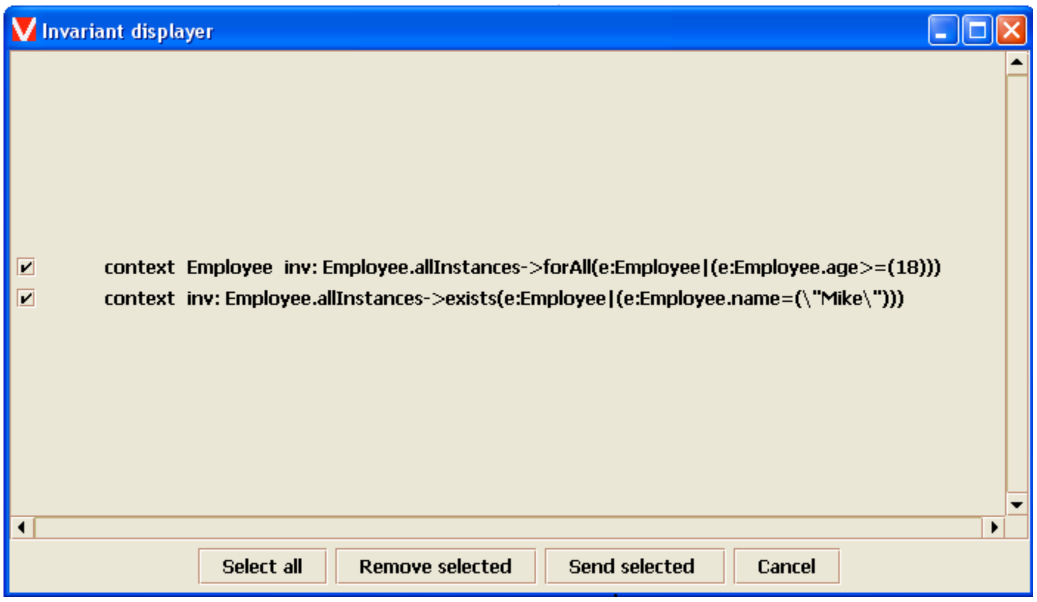


```
<integer>
<string>
<type>
Employee.allInstances
if(<Boolean>)then(<<type>:1>)else(<<type>:2>)fi
not(<Boolean>)
Project.allInstances
self:Employee
Set{<<type>>}->union(<Set[<type>]>)
Set{}
```

Figure 9.12: OCL Editor owned by the MOVA tool

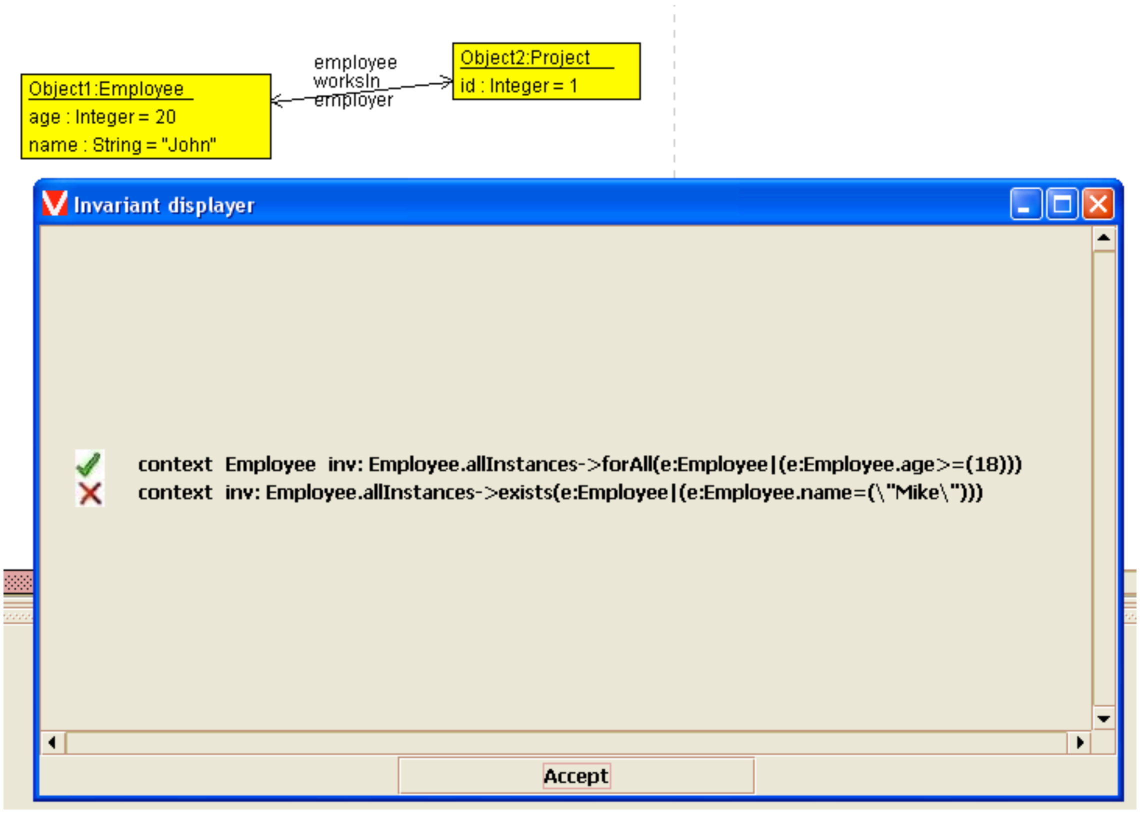
For example, if we select the *Employee.allInstances* alternative we can press the **->** button to obtain a list containing all the possibilities to use (e.g., forAll, exists, ..., and all collection operations).

The editor provides three buttons, arrow (->), dot (.) and Space to complete constraints. We only have to note that it is not possible to edit a constraint to add more contents (e.g., in a boolean expression to add another condition with an *and* operator). Nevertheless, this tool is excellent for people that are not used to work with OCL because it helps them to make correct constraints from the beginning.



☑ context Employee inv: Employee.allInstances->forAll(e:Employee|(e:Employee.age>=(18)))
☑ context inv: Employee.allInstances->exists(e:Employee|(e:Employee.name=(\"Mike\")))

Figure 9.13: Invariants in MOVA tool



✓ context Employee inv: Employee.allInstances->forAll(e:Employee|(e:Employee.age>=(18)))
✗ context inv: Employee.allInstances->exists(e:Employee|(e:Employee.name=(\"Mike\")))

Figure 9.14: Result of invariant evaluations within MOVA tool

At Figure 9.13 we can see the invariants written by the step-by-step editor. It is important to note that such invariants can be sent to the object diagram to be executed and evaluate the system. For to do this we have the *"Send selected"* button on this view.

Last step is to evaluate the constraints with the object instances. Pushing the evaluation button in the object diagram we obtain a window with all the constraints sent and an image, which precedes each of them. Such images denote if the invariant constraint evaluates to true (denoted with a green tick symbol) or false (denoted with a red cross), as we can see at Figure 9.14.

As a conclusion for the MOVA tool, it is important to emphasize that the way of writing invariant expressions is very interesting in order to learn how the OCL 2.0 works. Nevertheless, for those that are used to work with this constraint language it is more useful to write expressions directly by hand.

MOVA tool is being an essential tool for students from the Universidad Complutense de Madrid, so this is a proof of its value as learning software for future computer engineers.

9.2.4 IBM OCL PARSER

Among the first tools developed for OCL, OCL PARSER 0.3 by Jos Warmer (co-author also of the book about OCL that can be found at [WK03]) stands out.

It is an analyser written in Java and is limited to syntactic analysis and a partial checking of types. It allows the analysis of constraints included in UML models, but requires a specific format of the tool. "OCL Parser 0.3" is available inside IBM web at [IOPw].

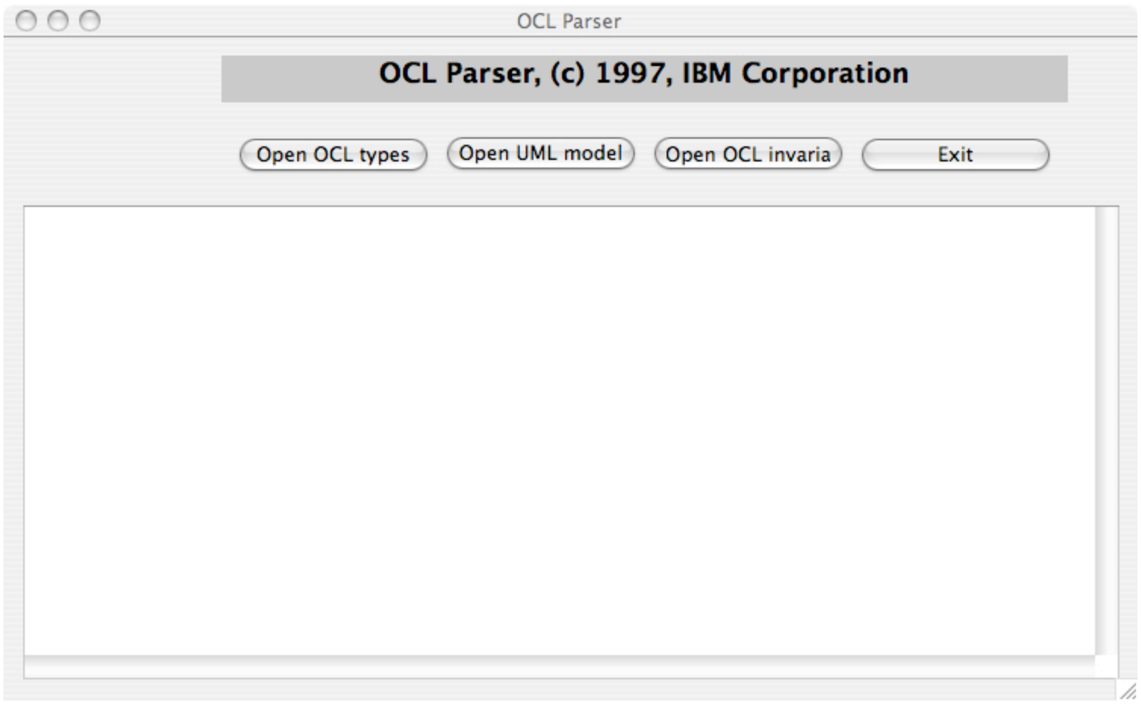


Figure 9.15: IBM OCL Parser 0.3

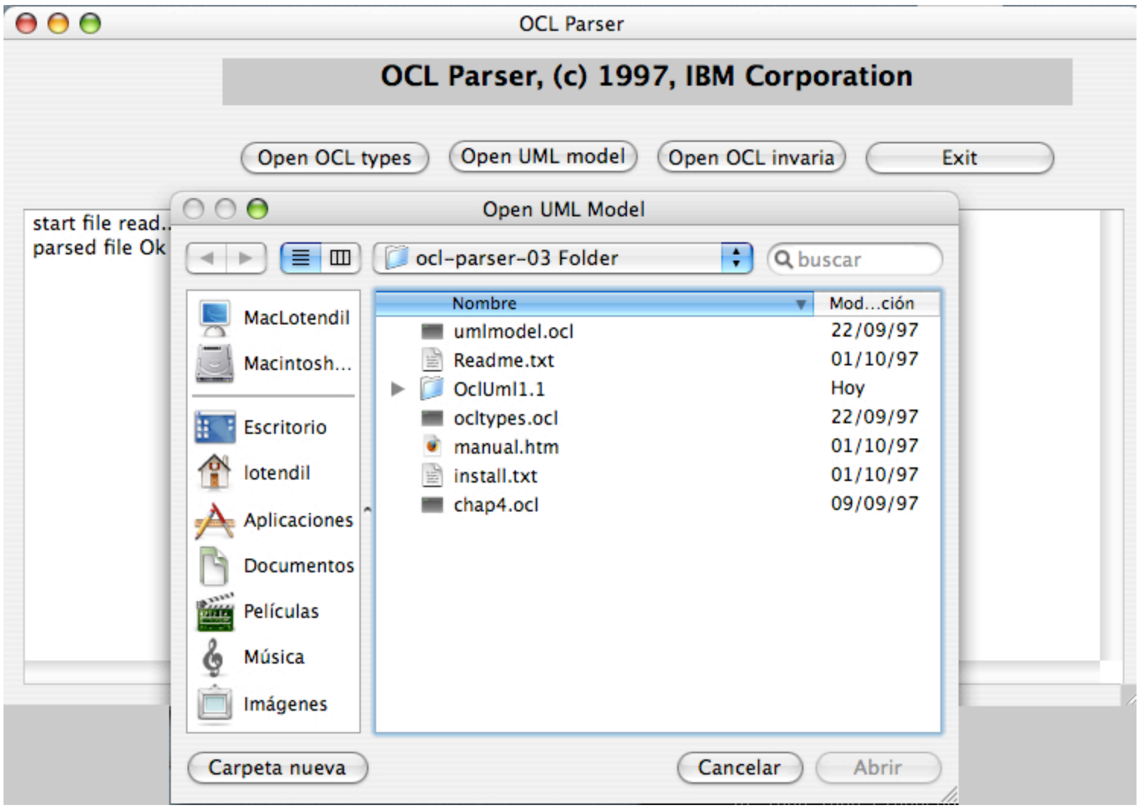
OCL Parser 0.3 provides support for UML and OCL versions 1.1. Once we open this tool we found a graphical interface like shown at Figure 9.15. A window comes up with four buttons:

- Open OCL types: this button opens a file containing all predefined OCL types. The OCL types file contains type, attribute and operation declarations, including subtype-supertype relationships of the predefined OCL types.

- Open UML model: this button is used to open a file that contains all information from a UML model that is needed in OCL expressions. The UML model file contains type, attribute and operation declarations, including subtype-supertype relationships.

- Open OCL invariants: this button is used to open and check a file containing OCL invariants on the types defined in the UML model file. The file will be read and all expressions will be checked.

- Exit: quits the application.



Nombre	Mod...ción
umlmodel.ocl	22/09/97
Readme.txt	01/10/97
OclUml1.1	Hoy
ocltypes.ocl	22/09/97
manual.htm	01/10/97
install.txt	01/10/97
chap4.ocl	09/09/97

Figure 9.16: Opening UML file into IBM OCL Parser 0.3

This tool uses a predefined format for both OCL and UML files because XMI was not implemented yet. Nevertheless, the format of these files is a simplified version of the XML language.

According to the user documentation within the release package of the IBM OCL Parser, version 0.3 is an incomplete implementation of OCL. It contains the following restrictions:

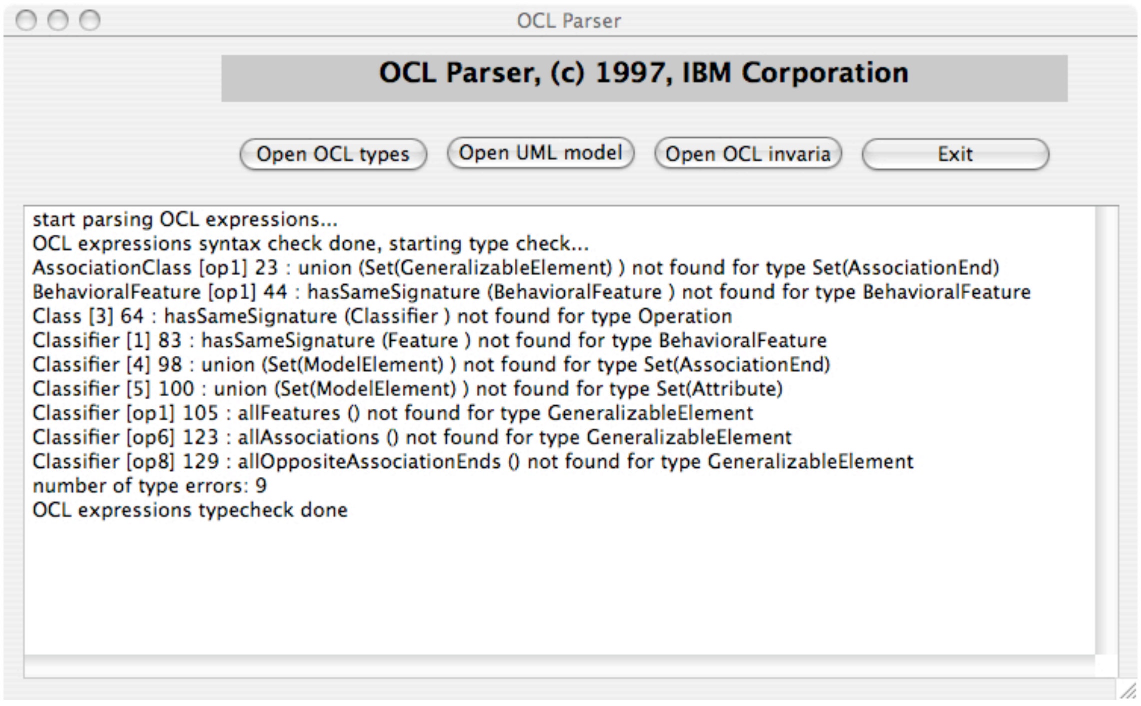
- There is no difference between an identical named attribute and operation without parameters. They are currently treated as being the identical.

- Type checking is not completely implemented. Therefore not all type errors are caught.

- Only invariants can be checked properly, pre and postconditions in the context of an operation are not handled yet.

- The source code is not documented well, but it is contained in the distribution.

- It has only been tested on the Windows95 platform, using SUN Microsystems JDK, version 1.1.



```
start parsing OCL expressions...
OCL expressions syntax check done, starting type check...
AssociationClass [op1] 23 : union (Set(GeneralizableElement) ) not found for type Set(AssociationEnd)
BehavioralFeature [op1] 44 : hasSameSignature (BehavioralFeature ) not found for type BehavioralFeature
Class [3] 64 : hasSameSignature (Classifier ) not found for type Operation
Classifier [1] 83 : hasSameSignature (Feature ) not found for type BehavioralFeature
Classifier [4] 98 : union (Set(ModelElement) ) not found for type Set(AssociationEnd)
Classifier [5] 100 : union (Set(ModelElement) ) not found for type Set(Attribute)
Classifier [op1] 105 : allFeatures () not found for type GeneralizableElement
Classifier [op6] 123 : allAssociations () not found for type GeneralizableElement
Classifier [op8] 129 : allOppositeAssociationEnds () not found for type GeneralizableElement
number of type errors: 9
OCL expressions typecheck done
```

Figure 9.17: Checking OCL into IBM OCL Parser 0.3

IBM OCL Parser 0.3 development seems to be stopped, because there are no changes since its last release version in 1997. Nevertheless, we have the source code available with the release package.

9.3 SUMMARY

Once we have seen a brief explanation of a set of tools that have the OCL as an important part of their functionality, we are able to affirm that they contain some similarities with our processor, like usage of the XMI format, parsing and instantiation of constraints, and error handling.

Nevertheless, some of these tools also have differences with our processor like usage of earlier versions of UML or OCL, or treatment of only a subset of these languages.

At next chapter we will select one of them in order to be the basis of our processor's design and development phases according to a formal process.

DECISION MAKING PROCESS 10

10 DECISION MAKING PROCESS

10.1 INTRODUCTION

It is possible to define human beings as decisional machines. Along the day we are used to make lots of decisions that affect to our behaviour or way of life. To make these decisions our brain does a process in which a set of alternatives are evaluated in order to choose such one that (sometimes) seems to be more reasonable.

There exist some decisions that are hard to make according to the consequences of their alternatives. For those problems we can use a formal method called multi-criteria decision analysis that can help us in order to convince us in that we are choosing the best alternative in a decision between different ones.

Along this chapter we will apply this process to the problem of choosing the best OCL tool to be the basis of our processor of OCL expressions.

10.2 MULTI-CRITERIA DECISION ANALYSIS (MCDA)

The multi-criteria decision analysis (MCDA) process is a discipline aimed at supporting decision makers who are faced with making numerous and conflicting evaluations. MCDA aims at highlighting these conflicts and deriving a way to obtain a good solution in a transparent process.

Our MCDA process applied to our problem will be based in a multi-attribute analysis and evaluation for each alternative that can be chosen.

10.2.1 A DECISIONAL PROBLEM

First step is to define our problem as a decisional problem. We are trying to choose an existent software tool between a set that will be the basis of our development phase for to implement a processor of OCL 2.0 expressions, which have to be instantiated into the OCL 2.0 metamodel. Such expressions should also be checked according to an UML model in order to find possible errors and help users to solve them.

Furthermore, the chosen alternative should be easy to adapt to the Eina GMC modeling tool due to the metamodels in which to instantiate the constraints are owned by such tool.

It is obvious that the one that has to make the decision is who is writing this document, but such decision can also affect future users of the processor. Therefore to know what the users need is an important task that has influence in the decision making process.

10.2.2 ALTERNATIVES TO CHOOSE

Another important task is to know and understand the alternatives that are part of the decisional problem. In our case, the software tools that contain OCL functionalities are the alternatives to choose.

It is important to emphasize that we don't want to open a fight between these tools in order to decide which one is the best. We only want to decide which tool best fits to our processor in mind to become the basis of its development.

Therefore we are not deciding if one tool is better that other one. We are choosing an alternative that owns features that can be adapted to our processor or that can be extended in order to be useful in our development.

10.2.2.1 USE Tool

The UML-based Specification Environment (USE) is one of our alternatives and has the advantage of provide a complete set of features related with the OCL language. USE is able to check syntax and semantics of OCL expressions as well as to interpret them in order to make queries to the system.

The main disadvantages of this tool are that the format used to load and save UML diagrams is not a standard like XMI, and the UML supported within USE is a subset of the whole. Nevertheless USE is already in development, so future version will include more functionalities that could solve such lacks.

A brief description about USE was done at section 9.2.1 of chapter 9. A full explanation can be found in [USE07].

10.2.2.2 Dresden OCL2 Toolkit

Inside Dresden OCL2 Toolkit we found the OCL 2.0 parser subsystem as a component. It is another alternative in our decisional problem.

The disadvantage of this tool related to our processor's needs is that UML version supported is 1.5 instead of later 2.0. Despite, it could be possible to adapt the base version in order to support UML 2.0 version.

On the other hand, the OCL 2.0 parser subsystem has the advantage of provide parsing support for the whole OCL 2.0 and UML 1.5, therefore it does not support only a subset. Moreover, it uses XMI format similar than Eina GMC one to load and store models.

A brief description about this tool was done at section 9.2.2 of chapter 9. A full explanation can be found in [Kon05].

10.2.2.3 MOVA Project

The MOVA tool as member of the MOVA project is our third alternative that we have presented to be chosen as a basis for the development of our processor.

Its main disadvantage is that it is not possible to write OCL expressions directly without using its step-by-step OCL editor. This lack implies that experienced users could consider such tool a little discomfort.

Nevertheless, MOVA tool has been tested with students of software engineering with success, so it is a proof of its usability with people that has not sufficient experience with modeling and constraint languages.

A brief description about MOVA tool was done at section 9.2.3 of chapter 9. A full explanation can be found in [MOVw].

10.2.2.4 IBM OCL Parser

The IBM OCL Parser is one of the first tools supporting OCL. It is the last alternative that will be studied in our multi-criteria analysis process in order to be the basis for our tool.

Its disadvantages are that it seems to be unfinished and that does not support last OCL 2.0 language. It only holds versions 1.1 of UML and OCL. Furthermore, the IBM OCL parser does not instantiate OCL expressions into a metamodel. It only checks syntax and semantics of such expressions.

On the other hand, its main advantage is its simplicity that can make easy the task of understanding its source code, although it is not documented in precise detail.

A brief description about IBM OCL Parser was done at section 9.2.4 of chapter 9. This tool can be found in [IOPw].

10.2.3 Structuring the problem: Value Tree

Next step in this process is to find measurable values in order to act as guides to the correct decision. A value can be defined as a concept that matters and identifies our needs.

This section aims to create a structure that simplifies the analysis process. As you can imagine if you are an experienced user in computer science or mathematics, best simple structure to use is a tree. In our case, it will be a value tree containing all the important values according to our problem.

For each criterion or essential purpose we have to think about value concepts that could measure it in order to create a hierarchical structure in tree form.

Remembering our problem, we need a tool that supports both UML and OCL languages in their 2.0 versions. In addition, such tool should check OCL expressions in order to verify syntax and semantics. A type conformance analysis is another important feature that cannot be forgiven.

We are also interested in a tool that supports XMI as model interchange format, and that instantiates OCL expressions into the OCL metamodel.

Furthermore, it is important to select a tool that provides documentation for its users and also more technical information, like design and implementation manuals or a well-structured source code with useful comments.

Finally, we don't have to forgive that such tool should be easy to use for both experienced and inexperienced users.

With such wish list we have to be able to make a value tree that will be the basis of this analysis process from now on. Therefore, to continue we will show the final list of value concepts and the tree-form structure.

- UML and OCL versions supported.
- UML model interchange format supported.
- OCL verification: syntax and semantics, and type conformance.
- OCL execution or interpretation.
- Usability of experienced and inexperienced users.
- User documentation provided.
- Technical documentation provided.
- Quality of source code.

Splitting and aggregating these previous value concepts we are able to present the hierarchical structure of the Figure 10.1.

It is important to emphasize that the information has been structured through three main nodes: features, documentation and usability. We think that these are the three pillars that contain all the other nodes inside the tree hierarchy.

As descendants of features we find the related nodes to the UML and OCL languages supported for our tools. Both contain information about version, instantiation, interchange format supported and all those concepts that we defined as concepts with an important role in the decision making process.

Documentation and usability nodes also contain their respective descendants with all the information required before.

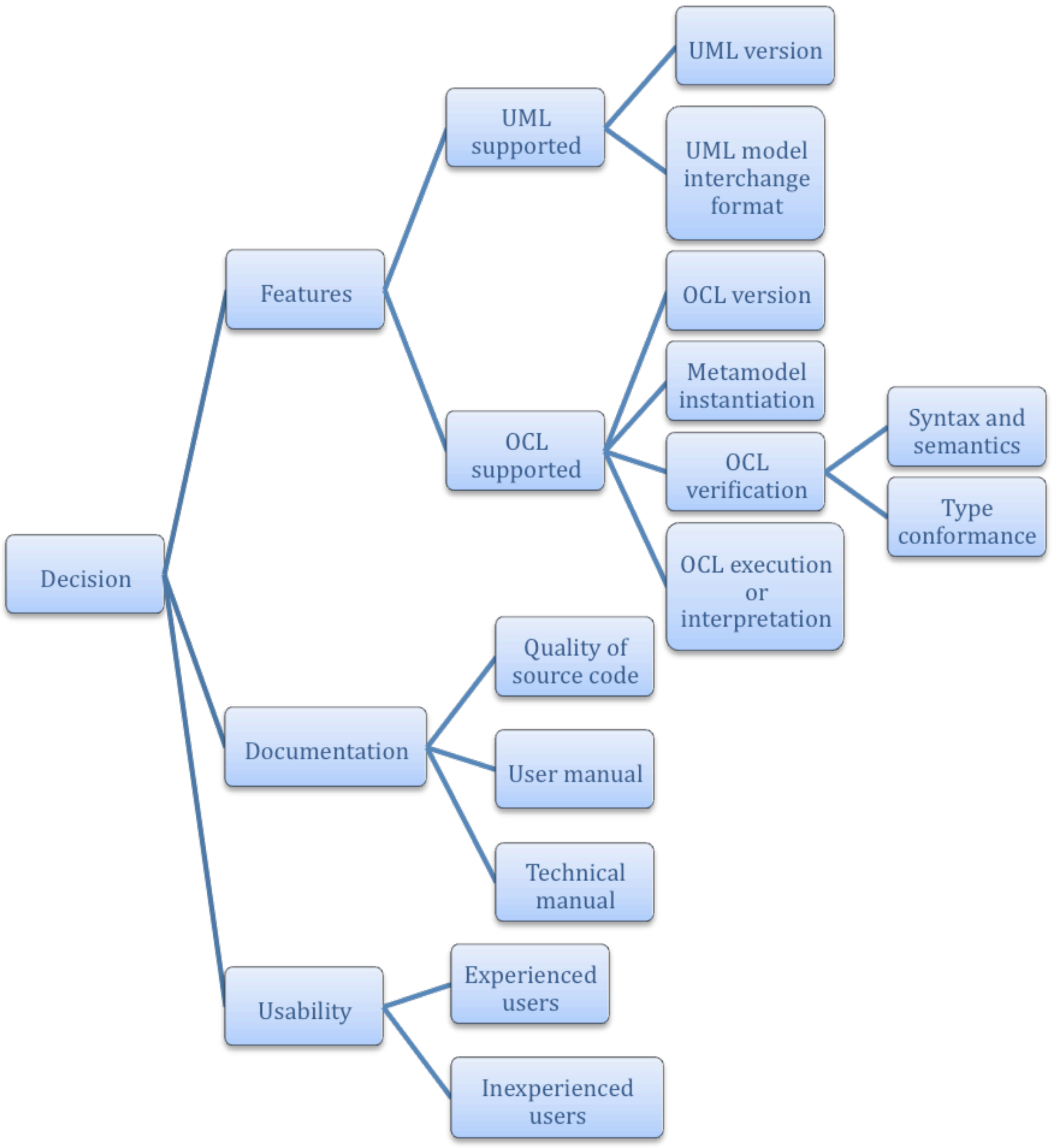


Figure 10.1: Tree value with value concepts that are essential in our decision analysis

At next section we will explain each node in detail in order to make a valuation of these value concepts that will be useful when evaluating every alternative that has possibility of to be chosen.

To do this qualifying process we will use the SMART method that will be explained before it.

10.2.4 ATTRIBUTE VALUATION: SMART METHOD

The Simple Multi Attribute Rating Technique, also known as SMART method, was firstly introduced by Edwards in 1977. The version applied at this chapter was explained during my classes on decision making and project management on business [PDGPE] at Barcelona School of Informatics.

The purpose of the SMART technique is to value the importance of each node of the value tree. This task consists on assign a weight to each attribute taking care that each group of siblings must have a total weight of 100%.

First of all, we have to start with the root attributes group, valuate them, and then continue this process with the descendant groups.

- Steps in the SMART process:

  1. For each group of attributes, sort them according to their importance
  2. Compare the importance of each attribute with respect the less important one.
  3. Transform importance values into weights.

In our value tree we can distinguish 7 different groups of attributes as we can see at Figure 10.2. Next to it we will apply this process to each group in order to obtain the correct weights of each leaf node.

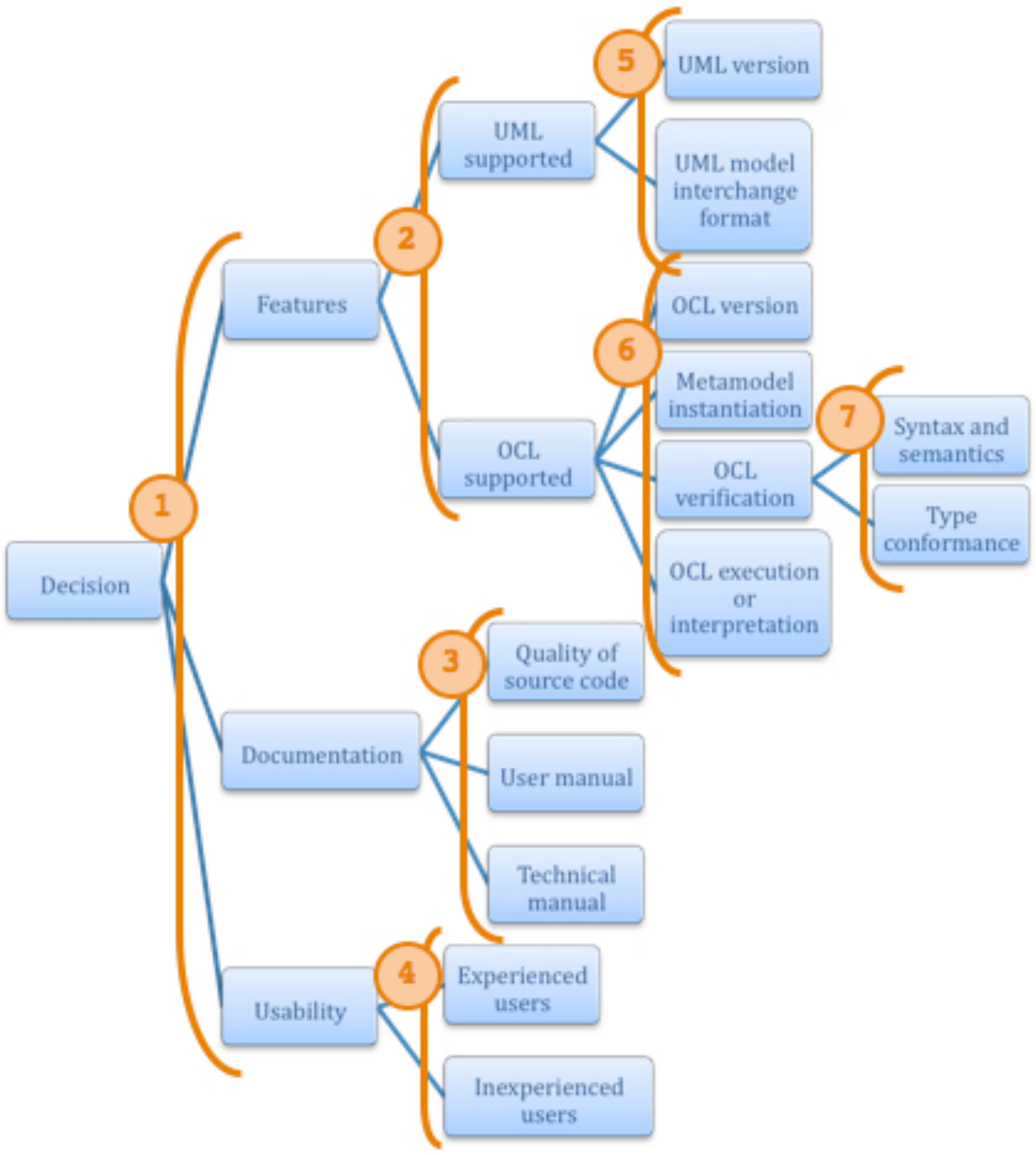


Figure 10.2: Value tree with groups

10.2.4.1 Group 1

This group contains the three root nodes about features, documentation and usability. As we indicated before, first step of the SMART technique is to sort these value concepts according to their importance. At next example we will see the result of this ordering.

Features = Documentation > Usability

We think that according to our needs, it is more important a tool with good and well-documented features than another one that has a good usability, because this last characteristic could be obtained by ourselves doing a good development although our base tool didn't have such characteristic.

Example 10.1: Ordering first group of value concepts

Then, next step is to compare the importance degree of each value concept with respect the others, and finally, convert importance values into weights.

Less important concept: Usability = 1

How many times is more important Documentation than Usability? 2

How many times is more important Features than Usability? 2

Weight(a) = importance(a) / Σ importance

Weight of Usability: 1/(2+2+1) = 0.2 ~ 20%

Weight of Documentation: 2/(2+2+1) = 0.4 ~ 40%

Weight of Features: 2/(2+2+1) = 0.4 ~ 40%

Example 10.2: Weights of group 1 members

Therefore, once we have done the weight calculation for this first group applying the SMART technique, we only have to repeat it with the other groups.

10.2.4.2 Group 2

Within second group there are only two nodes representing the UML and OCL languages supported, so here it is not necessary to apply the SMART method. We think that both concepts should have the same weight, so due to the sum of such weights must be 100%, each one of these concepts has a weight of 50%.

10.2.4.3 Group 3

Within the Documentation node we found three nodes: Quality of source code, user manual and technical manual. Then we will apply the SMART method first step consisting in ordering such nodes according to their importance degree.

Source code = Technical manual > User manual

We think that according to our needs, it is more important a tool with a good-structured source code and a technical manual where all design and implementation decisions are explained in detail than another one that has a good user manual, because this last characteristic will be not as important as others in the development phase of our processor.

Example 10.3: Ordering third group of value concepts

Then, next step is to compare the importance degree of each value concept with respect the others, and finally, convert importance values into weights.

Less important concept: User manual = 1

How many times is more important a Technical manual than a User manual? 3

How many times is more important a well-documented Source code than a User manual? 3

Weight(a) = importance(a) / Σ importance

Weight of User manual: 1/(3+3+1) = 0.143 ~ 14%

Weight of Technical manual: 3/(3+3+1) = 0.428 ~ 43%

Weight of Source code: 3/(3+3+1) = 0.428 ~ 43%

Example 10.4: Weights of group 3 members

10.2.4.4 Group 4

Within fourth group there are only two nodes representing the usability for experienced and inexperienced users of the tool, so here it is not necessary to apply the SMART method. We think that both concepts should have the same weight, so due to the sum of such weights must be 100%, each one of these concepts has a weight of 50%.

10.2.4.5 Group 5

Within 5th group there are only two nodes representing the UML version supported by the tool being evaluated, and the kind of interchange format for to represent UML models, so here it is not necessary to apply the SMART method. We think that the interchange format concept is more important here because our processor will work with XMI. On the other hand, if UML version is earlier than last 2.0 we think that to adapt it to new UML version could be easier than to adapt interchange format to XMI.

Therefore, we our decision is to add a 40% weight for UML version and a 60% for the interchange format node.

10.2.4.6 Group 6

Within the OCL supported node we found four nodes: OCL version, OCL instantiation into a metamodel, OCL verification and OCL execution or interpretation.

Then we will apply the SMART method first step consisting in ordering such nodes according to their importance degree.

Metamodel instantiation = OCL verification > OCL version > OCL execution

We think that according to our needs, it is more important a tool that instantiates OCL into a metamodel because it is the main purpose of our processor. This concept has the same importance than the verification of OCL due to we need a tool that will be able to find errors in OCL expressions.

Finally, we think that OCL version supported is more important than if the tool executes OCL expression because we are more interested in tools that support OCL 2.0 than in tools that are able to execute OCL expressions because it is not our purpose, although a tool with this feature will be well-valued.

Example 10.5: Ordering 7th group of value concepts

Then, next step is to compare the importance degree of each value concept with respect the others, and finally, convert importance values into weights.

Less important concept: OCL execution = 1

How many times is more important the OCL version supported than the execution of OCL? 2

How many times is more important a tool with OCL verification than the execution of OCL? 3

How many times is more important a tool that instantiates OCL expression than the execution of OCL? 3

Weight(a) = importance(a) / Σ importance

Weight of OCL execution: 1/(3+3+2+1) = 0.11 ~ 12%

Weight of OCL version: 2/(3+3+2+1) = 0.22 ~ 22%

Weight of OCL verification: 3/(3+3+2+1) = 0.33 ~ 33%

Weight of metamodel instantiation: 3/(3+3+2+1) = 0.33 ~ 33%

Example 10.6: Weights of group 3 members

10.2.4.7 Group 7

Within 7th group there are only two nodes representing if the tool being evaluated checks both syntax and semantics, and type conformance of OCL expressions, so here it is not necessary to apply the SMART method. We think that both concepts should have the same weight, so due to the sum of such weights must be 100%, each one of these concepts has a weight of 50%.

10.2.4.8 Final value tree

			Partial Weights	Leaf Weight
Features			40%	▨
UML supported			50%	▨
	UML version		*40%*	8,0%
	UML model interchange format		*60%*	12,0%
OCL supported			50%	▨
	OCL version		*22%*	4,4%
	Metamodel instantiation		*33%*	6,6%
	OCL verification		*33%*	▨
		Syntax and semantics	50%	3,3%
		Type conformance	50%	3,3%
	OCL execution or interpretation		*12%*	2,4%
Documentation			40%	▨
Quality of source code			43%	17,2%
User manual			14%	5,6%
Technical manual			43%	17,2%
Usability			20%	▨
Experienced users			50%	10,0%
Inexperienced users			50%	10,0%
				100%

Table 10.1: Weights of value concepts

This table contains the total weights of the leaf nodes. They are calculated multiplying all percentages from the root to each terminal node.

For example, we can see that the weight of type conformance node is computed as follows:

Partial Weight(Features) = 40%
Partial Weight(OCL supported) = 50%
Partial Weight(OCL verification) = 33%
Partial Weight(Type conformance) = 50%

Therefore,
Total Weight(Type conformance) = 40% x 50% x 33% x 50% ⇒ 3,3%

Example 10.7: Calculation of total weight

10.2.5 Evaluation of alternatives

Once we have all weights for our terminal nodes of the value tree, next step is to define how to evaluate each alternative through these nodes.

The aim of this section is to complete a table for each alternative composed by the terminal nodes of the value tree. For each terminal node there is a cell in the table that must contain the evaluation of the alternative for such concept. For example, inside the cell of the concept *UML version* we could write *2.0* representing that such tool supports UML 2.0 version.

In the next table we can understand which are the values that can be placed for each terminal node of the value tree. Next to it we will show one table for each alternative that participates in our decision problem.

	Total weigth	Allowed values
UML version	8,0%	version number
UML model interchange format	12,0%	kind of format
OCL version	4,4%	version number
Metamodel instantiation	6,6%	Yes or no
Syntax and semantics	3,3%	Syntax, semantics, both or none
Type conformance	3,3%	Yes or no
OCL execution or interpretation	2,4%	Way of doing this feature
Quality of source code	17,2%	0 - 10
User manual	5,6%	0 - 10
Technical manual	17,2%	0 - 10
Experienced users	10,0%	0 - 10
Inexperienced users	10,0%	0 - 10
	100,0%	

Table 10.2: Allowed values for the terminal nodes of the value tree

So, our decision will be determined with the values of these 12 concepts. Note that last five concepts of Table 10.2 are evaluated using numbers in a 0-10 quality scale where 0 is lower value and 10 is upper value.

10.2.5.1 USE tool

Next table represents the evaluation of USE tool for the current terminal nodes of our value tree.

It is important to note that it has a lower value in quality of technical manual concept due to in USE web [USEw] not exists a technical manual explaining the design and implementation in detail of such tool.

Alternative: **USE Tool**

	Total weigth	Evaluation
UML version	8,0%	2.0
UML model interchange format	12,0%	own format
OCL version	4,4%	2.0
Metamodel instantiation	6,6%	No
Syntax and semantics	3,3%	Both
Type conformance	3,3%	Yes
OCL execution or interpretation	2,4%	OCL directly interpreted
Quality of source code	17,2%	8
User manual	5,6%	8
Technical manual	17,2%	4
Experienced users	10,0%	8
Inexperienced users	10,0%	6
	100,0%	

Table 10.3: Evaluation of USE tool

10.2.5.2 Dresden OCL2 Toolkit

Next table represents the evaluation of Dresden OCL2 Toolkit for the current terminal nodes of our value tree.

Alternative: **Dresden OCL2 Toolkit**

	Total weigth	Evaluation
UML version	8,0%	1.5
UML model interchange format	12,0%	XMI
OCL version	4,4%	2.0
Metamodel instantiation	6,6%	Yes

Syntax and semantics	3,3%	Both
Type conformance	3,3%	Yes
OCL execution or interpretation	2,4%	Generating code
Quality of source code	17,2%	8
User manual	5,6%	5
Technical manual	17,2%	9
Experienced users	10,0%	8
Inexperienced users	10,0%	5
	100,0%	

Table 10.4: Evaluation of Dresden OCL2 Toolkit

It is important to emphasize that Dresden tool provides support to interpret OCL through generation of Java code, but such feature has a lower quality than USE tool, so this value will be lower than in USE.

10.2.5.3 MOVA Project

Next table represents the evaluation of MOVA tool for the current terminal nodes of our value tree.

Alternative: **MOVA Project**

	Total weigth	Evaluation
UML version	8,0%	2.0
UML model interchange format	12,0%	own format
OCL version	4,4%	2.0
Metamodel instantiation	6,6%	No
Syntax and semantics	3,3%	Both
Type conformance	3,3%	Yes

OCL execution or interpretation	2,4%	OCL interpreted
Quality of source code	17,2%	Source code not available
User manual	5,6%	8
Technical manual	17,2%	4
Experienced users	10,0%	7
Inexperienced users	10,0%	9
	100,0%	

Table 10.5: Evaluation of MOVA Project

The main problem of MOVA project is that source code is not directly available for everybody. Therefore we cannot evaluate in a positive way this decision because we need the code to adapt it to our processor. In spite of it, we think that we could obtain this source code making a request to MOVA developers.

It is also important to note that within this tool OCL is interpreted and executed but not directly as in USE. MOVA can only execute OCL placed as constraints of a model whereas USE can execute OCL expressions without a model as for example operations with collection of Integers.

10.2.5.4 IBM OCL Parser

Next table represents the evaluation of IBM OCL Parser for the current terminal nodes of our value tree.

Alternative: **IBM OCL Parser**

	Total weigth	Evaluation
UML version	8,0%	1.1
UML model interchange format	12,0%	own format
OCL version	4,4%	1.1

Metamodel instantiation	6,6%	No
Syntax and semantics	3,3%	Both
Type conformance	3,3%	Yes
OCL execution or interpretation	2,4%	No
Quality of source code	17,2%	7
User manual	5,6%	4
Technical manual	17,2%	3
Experienced users	10,0%	6
Inexperienced users	10,0%	4
	100,0%	

Table 10.6: Evaluation of IBM OCL Parser

The main problem of this tool is that support older version of both UML and OCL languages, so it will penalize this alternative.

10.2.6 Normalization

In order to evaluate each alternative we have chosen a value for each attribute that is important in this analysis process. But these attributes (or terminal nodes of the value tree) are defined in different scales.

The normalization process consists on transforming the values of the attributes to a unique scale in order to aggregate them and then obtain the total value of each alternative.

We will use a 0 - 100 scale, so we will describe what we have to do with each attribute for to obtain its normalized values.

10.2.6.1 UML version

This attribute represents the UML version supported for the tool that is evaluated. Its values are 1.1, 1.5 or 2.0.

We have followed the next rule to normalize such values:

Normalization of attribute UML version:

UML version	⇒	Normalized value
1.1	⇒	30
1.5	⇒	60
2.0	⇒	90

We have chosen these scale change to indicate that tools with an earlier version of UML are better valued.

Example 10.8: Normalization of UML version

10.2.6.2 UML model interchange format

This attribute represents which is the format used to load, store and share UML models by the tool that is evaluated. Such formats are XMI, which is the better valued, and a format owned by each tool, i.e., not standard format.

Normalization of attribute model interchange format:

Interchange format	⇒	Normalized value
own format	⇒	40
XMI	⇒	100

Example 10.9: Normalization of UML model interchange format

10.2.6.3 OCL version

This attribute represents the OCL version supported for the tool that is evaluated. Its values are 1.1 or 2.0.

We have followed the next rule to normalize such values:

Normalization of attribute OCL version:

OCL version ⇒ Normalized value

1.1 ⇒ 30

2.0 ⇒ 90

We have chosen these scale change to indicate that tools with an earlier version of OCL are better valued.

Example 10.10: Normalization of OCL version

10.2.6.4 Metamodel instantiation

This attribute represents if a tool instantiates OCL expressions and UML elements in a metamodel. This feature is very important for us, so it has followed the next rule to do the normalization.

Normalization of attribute Metamodel instantiation:

supports instantiation? ⇒ Normalized value

No ⇒ 30

Yes ⇒ 100

We put 30 instead of 0 for the 'No' value in order to avoid large deviations between the two possibilities.

Example 10.11: Normalization of Metamodel instantiation

10.2.6.5 Syntax and semantics

This attribute represents if a tool checks if OCL expressions are well formed and use correctly UML elements of the models. This feature is normalized following the next rule.

Normalization of attribute Syntax and semantics:

checks syntax and semantics? ⇒ Normalized value

No ⇒ 0

Only Syntax ⇒ 30

Both ⇒ 100

Example 10.12: Normalization of Syntax and semantics

10.2.6.6 Type conformance

This attribute indicates if a tool checks if the types of different elements in OCL expressions are compatible. Such feature is essential for correct usage of operations and for a possible future interpretation of the language.

This feature is normalized following the next rule.

Normalization of attribute Type conformance:

checks type of OCL expression? ⇒ Normalized value

No ⇒ 30

Yes ⇒ 100

Example 10.13: Normalization of Type conformance

10.2.6.7 OCL execution or interpretation

This attribute indicates if a tool executes or interprets OCL expressions. Although this feature is not an objective for our processor of OCL expressions, tools providing this functionality will be compensated with a better value in this attribute.

This feature is normalized following the next rule.

Normalization of attribute OCL execution or interpretation:

value ⇒ Normalized value

OCL directly interpreted ⇒ 100

OCL interpreted ⇒ 75

Generating code ⇒ 50

No ⇒ 25

Example 10.14: Normalization of OCL execution or interpretation

10.2.6.8 Quality of source code

This attribute represents with a number from 0 to 10 the quality of the source code of the tool that is evaluated.

Normalization of attribute Quality of source code:

value ⇒ Normalized value

0-10 number multiplied by 10 to obtain a 0-100 scale

Example 10.15: Normalization of Quality of source code

It is important to note that the particular case of MOVA tool where the source code is not directly available with the release of the tool will be evaluated with a value of 20 in order to avoid large deviations between the values of this attribute.

10.2.6.9 User manual

This attribute represents with a number from 0 to 10 the quality of the user manual of the tool that is evaluated.

Normalization of attribute Quality of User manual:

value ⇒ Normalized value

0-10 number multiplied by 10 to obtain a 0-100 scale

Example 10.16: Normalization of User manual

10.2.6.10 Technical manual

This attribute represents with a number from 0 to 10 the quality of the technical manuals of the tool that is evaluated.

Normalization of attribute Quality of Technical manuals:

value ⇒ Normalized value

0-10 number multiplied by 10 to obtain a 0-100 scale

Example 10.17: Normalization of Technical manual

10.2.6.11 Experienced users

This attribute represents with a number from 0 to 10 the usability of the tool that is evaluated with respect to experienced users.

Normalization of attribute Experienced users:

value ⇒ Normalized value

0-10 number multiplied by 10 to obtain a 0-100 scale

Example 10.18: Normalization of Experienced users

10.2.6.12 Inexperienced users

This attribute represents with a number from 0 to 10 the usability of the tool that is evaluated with respect to inexperienced users.

Normalization of attribute Inexperienced users:

value ⇒ Normalized value

0-10 number multiplied by 10 to obtain a 0-100 scale

Example 10.19: Normalization of Inexperienced users

10.2.7 Aggregation of Values

This final process calculates the utility of each alternative according to both the weight of each attribute and the normalized value of them.

Adding the values that result from this process we obtain the final value of each alternative. At next example we will see how to calculate these values, also called weighed values.

How to calculate weighed value for UML version attribute for USE alternative:

Weight of UML Version attribute: 8.0%

Evaluation for USE Tool alternative: 2.0

Normalized value: 2.0 ⇒ 90

Weighed value = Weight x Normalized value = 8.0% x 90 = 7.2

Example 10.20: How to calculate weighed value for an attribute

Next step is to show the four tables for our four alternatives, including normalized values, weighed values, and the aggregation of these ones.

It is important to note that we can find the final evaluation value for each alternative inside these tables.

10.2.7.1 USE tool

This table shows the complete evaluation for the USE tool. Weighed values are calculated as explained at Example 10.20.

Alternative: **USE Tool**

	Total weigth	Evaluation	Normalized values	Weighed values
UML version	8,0%	2.0	90	7,2
UML model interchange format	12,0%	own format	40	4,8
OCL version	4,4%	2.0	90	3,96
Metamodel instantiation	6,6%	No	30	1,98
Syntax and semantics	3,3%	Both	100	3,3
Type conformance	3,3%	Yes	100	3,3
OCL execution or interpretation	2,4%	OCL directly interpreted	100	2,4
Quality of source code	17,2%	8	80	13,76
User manual	5,6%	8	80	4,48
Technical manual	17,2%	4	40	6,88
Experienced users	10,0%	8	80	8
Inexperienced users	10,0%	6	60	6
	100,0%		**TOTAL**	66,06

Table 10.7: Table of aggregated values for USE tool

It is important to note that this alternative has a utility value of 66.06, which is the addition of each attribute utility.

10.2.7.2 Dresden OCL2 Toolkit

This table shows the complete evaluation for the Dresden OCL2 Toolkit. Weighed values are calculated as explained at Example 10.20.

Alternative: **Dresden OCL2 Toolkit**

	Total weigth	Evaluation	Normalized values	Weighed values
UML version	8,0%	1,5	60	4,8
UML model interchange format	12,0%	XMI	100	12
OCL version	4,4%	2.0	90	3,96
Metamodel instantiation	6,6%	Yes	100	6,6
Syntax and semantics	3,3%	Both	100	3,3
Type conformance	3,3%	Yes	100	3,3
OCL execution or interpretation	2,4%	Generating code	50	1,2
Quality of source code	17,2%	8	80	13,76
User manual	5,6%	5	50	2,8
Technical manual	17,2%	9	90	15,48
Experienced users	10,0%	8	80	8
Inexperienced users	10,0%	5	40	4
	100%		**TOTAL**	79,20

Table 10.8: Table of aggregated values for Dresden OCL2 Toolkit

It is important to note that this alternative has a utility value of 79.20, which is the addition of each attribute utility.

10.2.7.3 MOVA Project

This table shows the complete evaluation for the MOVA Project. Weighed values are calculated as explained at Example 10.20.

Alternative: **MOVA Project**

	Total weigth	Evaluation	Normalized values	Weighed values
UML version	8,0%	2.0	90	7,2
UML model interchange format	12,0%	own format	40	4,8
OCL version	4,4%	2.0	90	3,96
Metamodel instantiation	6,6%	No	30	1,98
Syntax and semantics	3,3%	Both	100	3,3
Type conformance	3,3%	Yes	100	3,3
OCL execution or interpretation	2,4%	OCL interpreted	75	1,8
Quality of source code	17,2%	Source code not available	20	3,44
User manual	5,6%	8	80	4,48
Technical manual	17,2%	4	40	6,88
Experienced users	10,0%	7	70	7
Inexperienced users	10,0%	9	90	9
	100,0%		**TOTAL**	57,14

Table 10.9: Table of aggregated values for MOVA Project

It is important to note that this alternative has a utility value of 57.14, which is the addition of each attribute utility.

10.2.7.4 IBM OCL Parser

This table shows the complete evaluation for the IBM OCL Parser. Weighed values are calculated as explained at Example 10.20.

Alternative: **IBM OCL Parser**

	Total weigth	Evaluation	Normalized values	Weighed values
UML version	8,0%	1.1	30	2,4
UML model interchange format	12,0%	own format	40	4,8
OCL version	4,4%	1.1	30	1,32
Metamodel instantiation	6,6%	No	30	1,98
Syntax and semantics	3,3%	Both	100	3,3
Type conformance	3,3%	Yes	100	3,3
OCL execution or interpretation	2,4%	No	25	0,6
Quality of source code	17,2%	7	70	12,04
User manual	5,6%	6	60	3,36
Technical manual	17,2%	3	30	5,16
Experienced users	10,0%	6	60	6
Inexperienced users	10,0%	4	40	4
	100,0%		**TOTAL**	48,26

Table 10.10: Table of aggregated values for IBM OCL Parser

It is important to note that this alternative has a utility value of 48.26, which is the addition of each attribute utility.

10.2.8 Final decision

As we have seen along this chapter, Dresden OCL2 Toolkit and particularly OCL Parser subsystem within it is the best tool for to be the basis of our processor of OCL expressions.

At next image we can see in a graphical way the result values of the Multi Criteria Decision Analysis (MCDA) process applied.

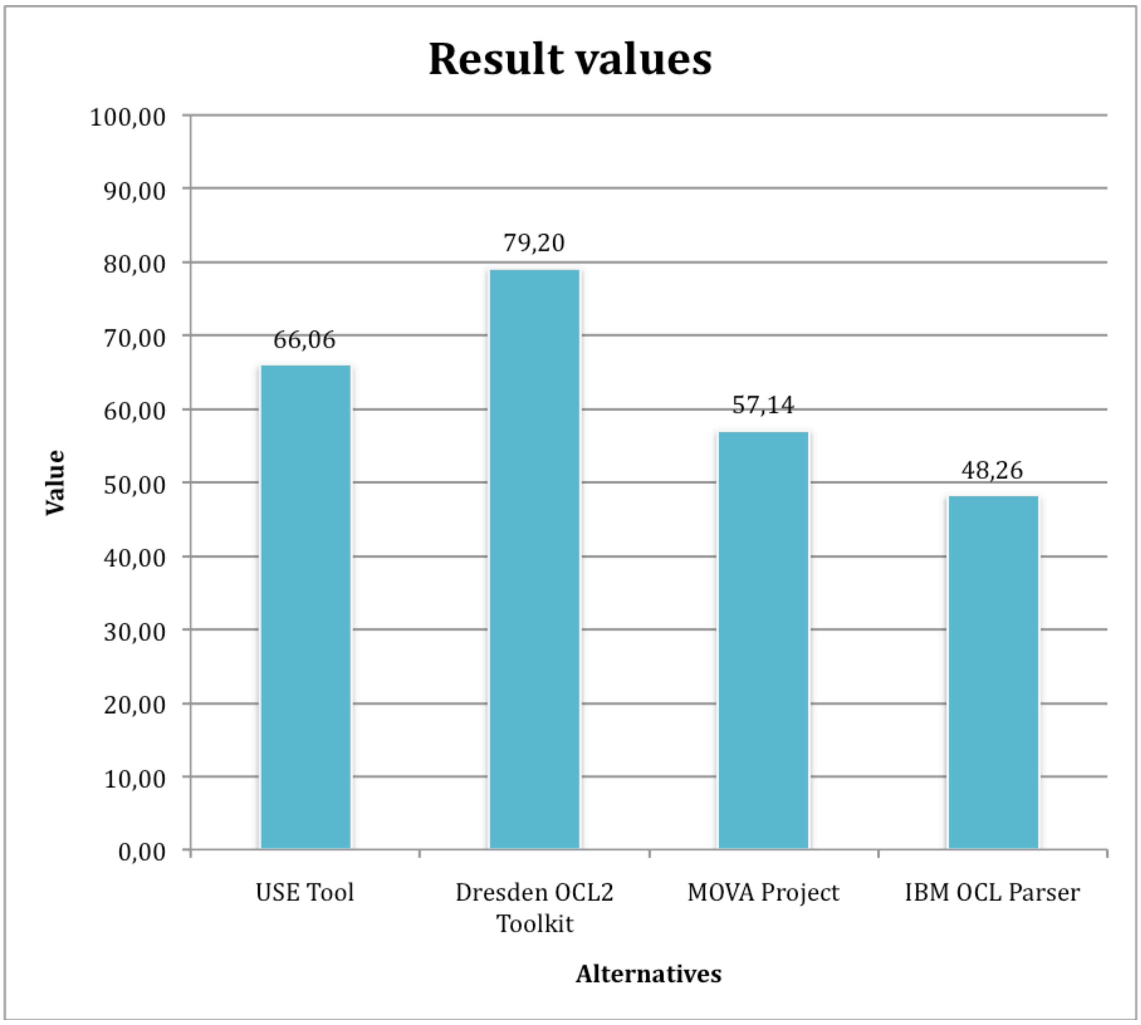


Figure 10.3: Result values of the decision making process

First position is for Dresden tool followed by USE, MOVA, and finally the IBM tool.

10.3 SUMMARY

In this chapter a formal analysis process has been used to determine which is the best alternative to choose in a complex decision.

There are a lot of situations where a formal process could be useful and respected by a majority instead of using only our feelings or sixth sense to make a decision.

In our case, a simple method is sufficient to obtain a good result and to make us able to decide which will be the OCL tool that we need to use as a basis for our design and implementation tasks of our processor of OCL expressions.

THE VISITOR PATTERN 11

11 THE VISITOR PATTERN

11.1 INTRODUCTION

Once we have chosen the existent OCL tool that will be the basis of the development phase of our project, in our case, the OCL 2.0 Parser Subsystem of the Dresden OCL2 Toolkit, to understand a basic design pattern that conforms the core of such tool becomes an essential job.

Therefore, along this chapter we will introduce the visitor pattern and its principal characteristics. If a more exhaustive explanation is needed, it can be found at [GH95].

11.2 A COMMON PROBLEM

Imagine you have different classes of products in an interactive web. For each product its cover image is displayed. Such products are DVDs, CDs, Books and Videogames, all derived from class Product (see Figure 11.1). Every time a user clicks an image, a method is called with the corresponding object as a parameter.

Such method needs to do a different job according to the class type of the object passed as a parameter. For example, it will print a message indicating the object class that was selected.

It could be implemented using if constructions and an operation that identifies the class type of an object (e.g., instanceof in Java programming language). One way to do this is as shown at Example 11.1.

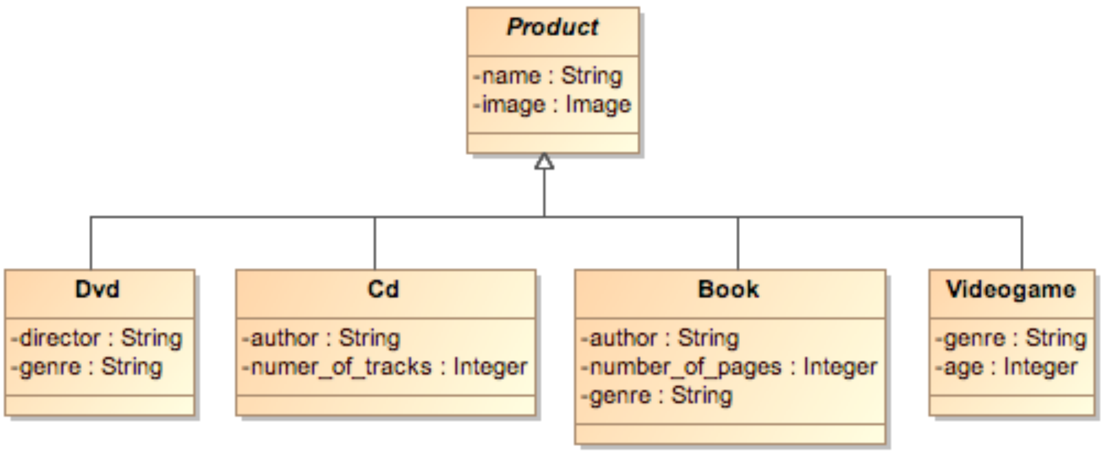


Figure 11.1: Hierarchical structure of Product objects

```
void process(Product p)
{
    if (p instanceof Dvd) {
       System.out.println(“You have chosen a DVD”);
    }
    else if (p instanceof Cd) {
       System.out.println(“You have chosen a CD”);
    }
    else if (p instanceof Book) {
       System.out.println(“You have chosen a Book”);
    }
    else if (p instanceof Videogame) {
       System.out.println(“You have chosen a Videogame”);
    }
}
```

Example 11.1: If construction and instanceof operation to process different objects

The solution proposed at Example 11.1 is *O(n)* at worst case because if we have *n* different classes of products, it could need *n-1* comparisons to find the correct case inside the if structure.

Therefore, we need a more efficient alternative to solve this problem. Another way to do this job but with a *O(1)* cost is to implement a method for each class that returns a number identifying each kind of product.

```
void process(Product p)
{
    switch (p.identifier())
    {
      case Dvd.IDENTIFIER:
        System.out.println(“You have chosen a DVD”);
        break;
      case Cd.IDENTIFIER:
        System.out.println(“You have chosen a DVD”);
        break;
      case Book.IDENTIFIER:
        System.out.println(“You have chosen a DVD”);
        break;
      case Videogame.IDENTIFIER:
        System.out.println(“You have chosen a DVD”);
        break;
    }
}
// Every class has a new attribute and a getter method as follows:
public static final int IDENTIFIER = x;  //where x is a number
public int identifier()
{
    return IDENTIFIER;
}
```

Example 11.2: Identifiers usage to identify each kind of product

However, since this solution delegates the control and responsibility of maintaining the identifiers to the programmer, it becomes error-prone because exists the possibility of having repetitions in the numbering of the identifiers.

Before explaining the correct solution using the visitor design pattern, we must introduce the concept of double dispatch mechanism.

11.3 DOUBLE DISPATCH

Double dispatch mechanism is useful in situations where the result of a computation depends on the run-time types of the objects that are part of it. It consists in a mechanism that sends a function call to different concrete methods depending on the run-time types of the objects involved in the call.

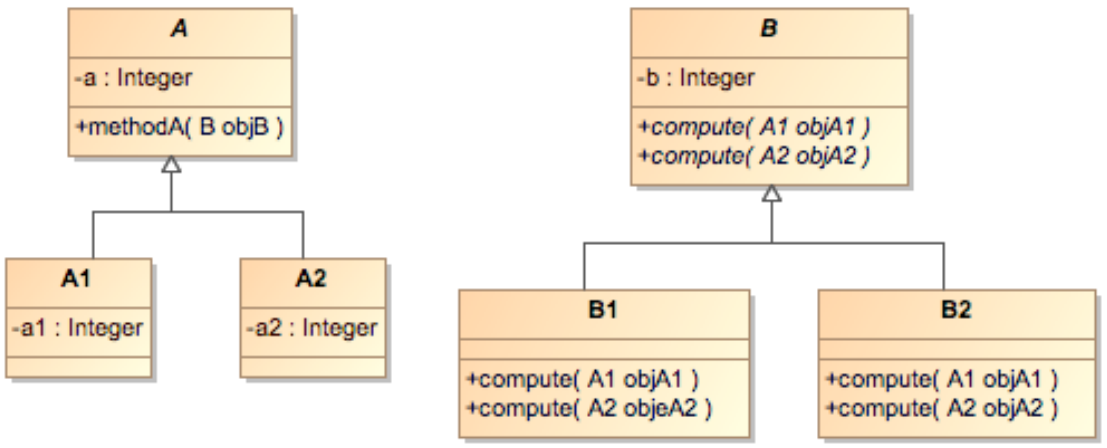


Figure 11.2: Object structure where to apply double dispatch mechanism

If we are in a situation similar than shown at Figure 11.2, we can instantiate objects of type A1 or A2, but not of type A because it is an abstract class. Similar behaviour happens with B structure.

Double dispatch mechanism: a way of send a function call to different concrete methods in two steps depending on the run-time types of the objects involved in the call.

Definition 11.1: Double dispatch mechanism

The A hierarchy represents a common object structure, and the B one represents the classes that contain methods to work over that hierarchy.

However, we can wrap a concrete class inside an abstract class as follows:

```
A mya1 = new A1(); // mya1 wraps and object of type A1
A mya2 = new A2(); // mya2 wraps and object of type A2
```

Example 11.3: Wrapping concrete instances into an abstract object

This case is correct because a superclass can be instantiated with a subclass instance. Then, we can work with A objects that internally are A1 or A2 objects.

Now, if we describe the methodA inside abstract class A as follows, we are able to introduce the double dispatch concept with an example.

We define `methodA` ***in abstract class A as follows:***

```
void methodA (B objB)
{
    objB.compute(this);
}
```

Example 11.4: Double dispatch mechanism

Since in B there are two methods named "compute" with different parameters, i.e., one with a A1 parameter and other with a A2 parameter, the method that will be called is determined by the run-time type of the A object.

The double dispatch mechanism is known with this name because of this double call. At Example 11.4 the outer call is needed to determine the type of the class that calls it, and the inner call determine the type of the parameter in the outer call to know which is the inner method because it is called over the outer parameter object, i.e., such method belongs to it.

We will see this behaviour in detail at following example. Its important to note that this double call mechanism is faster and avoids to use identifiers or other constructions that can introduce errors.

Our purpose is to maintain a structure of objects and another one where to declare the working classes, that is, the classes that have the responsibility to do the work needed over the object structure.

If we declare:

```
A mya1 = new A1();
A mya2 = new A2();
B myb1 = new B1();
B myb2 = new B2();
```

we have now two A objects and two B objects and we can call the following:

```
myA1.methodA(myb2); // that internally calls myb2.compute(this);
```

this call will do:

1- `myA1` ***is a reference to a*** `A1` ***instance, so at run-time this is known So, the call method is*** `A::methodA` ***that is common for both*** `A1` ***and*** `A2` ***instances***

2- ***Internally*** `myb2.compute(this)` ***is called, where*** `myb2` ***is a reference to a*** `B2` ***instance. So, it will use one of the two*** `compute` ***methods declared into*** `B2` ***class. Concretely, it will be decided once the type of this is evaluated. Then, we are calling the*** `compute` ***method inside the*** `methodA` ***called for*** `myA1`***, which is the*** `'this'` ***instance, and we agree that*** `myA1` ***is of*** `A1` ***type. Therefore, the*** `compute` ***method of*** `B2` ***used is the one that has a*** `A1` ***object as a parameter, i.e., the*** `B2::compute(A1)`***.***

Example 11.5: Double dispatch explained

With this kind of double calls we can separate the hierarchy of objects from the algorithms that work over them, as we said before.

Now, at next section we will study how to apply the double dispatch as a core of the visitor pattern using the previous example of products.

11.4 VISITOR PATTERN STRUCTURE

We have defined the visitor pattern structure as a way of separating an algorithm from an object structure. At this section we will study how is the pattern structure to apply.

Visitor pattern: design pattern that allows adding methods to an existing object hierarchy without modifying it and separating such object structure from the algorithms that works over it.

Definition 11.2: Visitor pattern

Fist step is to create two interfaces that will identify the two different parts of the hierarchy to separate. The Visitable interface determine the accept method that must be included at every concrete class extending it, and the Visitor interface introduces the visit method. At each subclass in the Visitor hierarchy must appear one visit method for every concrete object of the Visitable hierarchy.

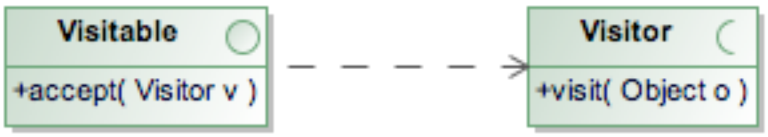


Figure 11.3: Visitor interfaces

Once we have a structure like in Figure 11.3, it is the moment to derivate it with our previous product hierarchy. Then, at Product class implementing Visitable interface we must complete the accept method applying the double dispatch explained before.

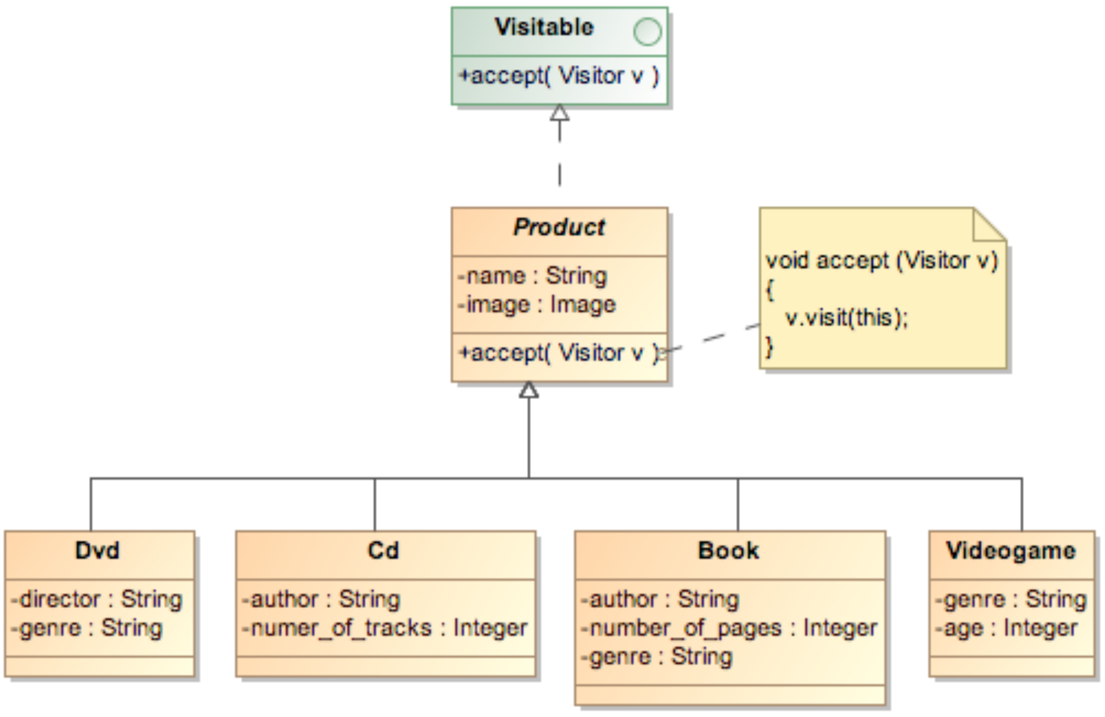


Figure 11.4: Visitable hierarchy with double dispatch

Then, at Visitor hierarchy we must create an abstract class with a visit method for every concrete Visitable object in order to provide a different method with the needed algorithm for every type of product.

At Figure 11.4 we can observe that the accept method is implemented inside the abstract class Product, so every subclass of it will have this method internally and it is not necessary to redefine it for each kind of Product.

Therefore, the only change to make respect to the previous object hierarchy is to add the Visitor interface defining the accept method and implement it into the Product class.

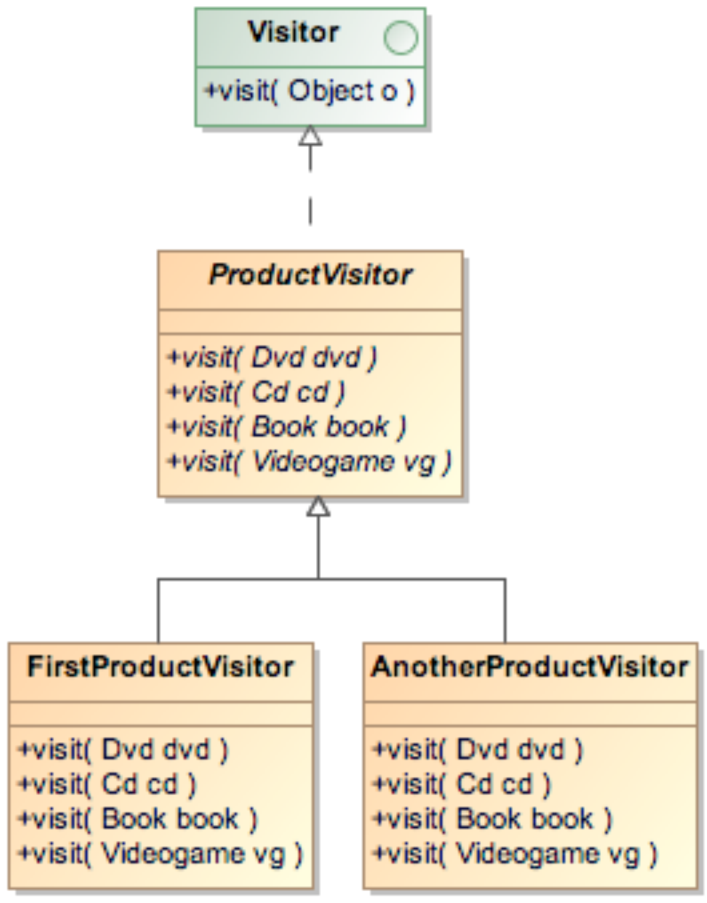


Figure 11.5: Visitor hierarchy with two extended visitor classes

Therefore, when the double dispatch mechanism is working, to find the correct method to call according to the run-time class of the caller object will be successful because a method for every object will exist.

Once we have our complete hierarchy, to process every product is very simple. For each kind of product we have to call the accept method, which is the first step of the double dispatch mechanism, with an instance of a concrete visitor. At second step, that is, calling the visit method of such visitor instance, the correct visit method is chosen according to the class type of the product where the accept method is called.

At next example of calling this behaviour is explained:

> ***Being p a Product of any concrete class and myVisitor an instance of FirstProductVisitor, we have to apply the double dispatch as follows:***
> `p.accept(myVisitor);//that internally calls myVisitor.visit(this);`
> ***As we have seen before, the*** visit ***method called is selected according to the type of the 'this' instance, which is the same*** p. ***For example, if*** p ***is a*** Dvd ***instance, the called method will be*** FirstProductVisitor::visit(Dvd dvd)

Example 11.6: Another double dispatch example

Therefore, the visitor pattern core is the double dispatch mechanism as we said earlier.

In order to understand the behaviour of this pattern definitively, we will introduce another example of problem solved applying such design pattern.

11.5 THE ROUTER CONFIGURATOR EXAMPLE

As an example of the visitor pattern, at this section we will study how to apply this pattern to a real case. Imagine that we need to develop a driver for every router model and operating system that allows each router to be configured. A short view of the object structure can be found at Figure 12.6.

We know that there exists a huge range of routers and a lot of different operating systems. To add a concrete method for each operating system configuration settings inside every router instance is not a good solution.

Therefore, to apply the visitor pattern in order to separate the object structure from the algorithms that configure each kind of router can be a good chance.

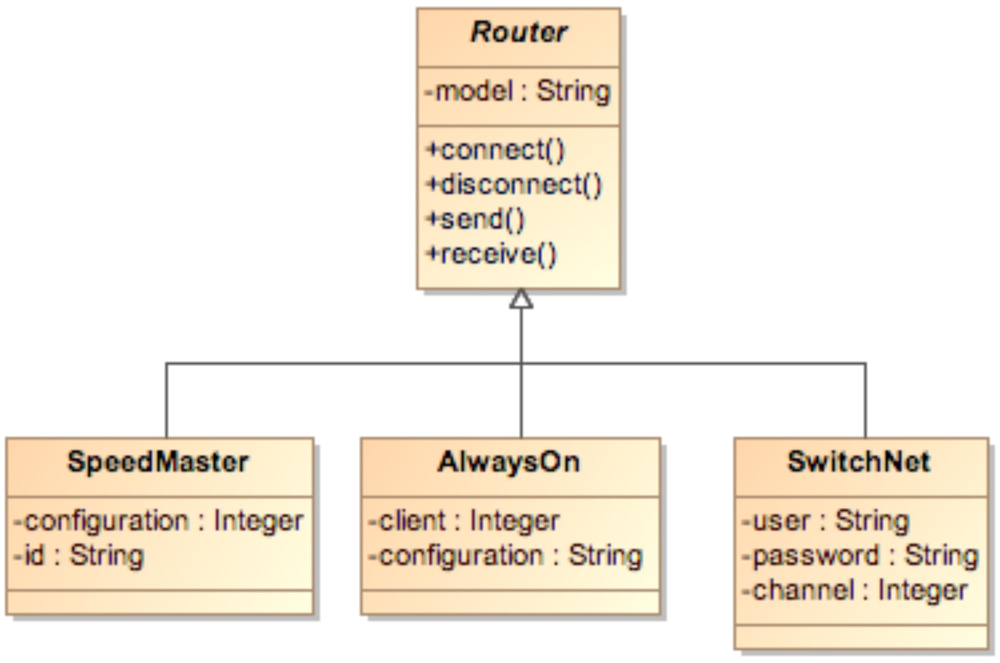


Figure 11.6: Basic structure of the router configurator example

As we can see, there are three different drivers in the example that implements the generic methods described at Router abstract class, one that drives a SpeedMaster router, another that drives a AlwaysOn router, and a third that drives the SwitchNet router.

We need to add a method to every router driver that puts the configuration parameters according to the operating system, but the object structure cannot be changed. Our aim is to avoid putting a configuration method in every driver for each operating system.

Applying the visitor pattern to our object structure results in an organization like shown at Figure 11.7.

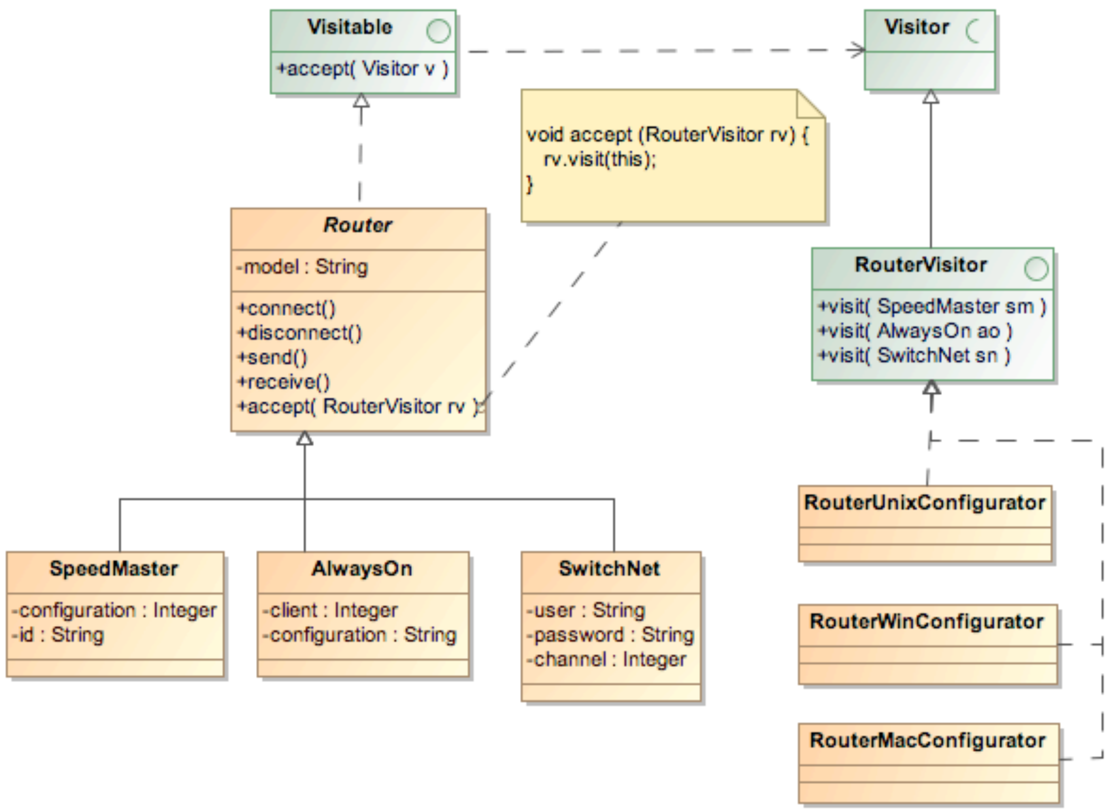


Figure 11.7: Visitor pattern applied to router configurator example

Is important to note that our job is to implement the custom visit methods of every class extending RouterVisitor. At image before, there are three router configurator classes to implement, one that makes the configuration of the modems for a Unix system, another for a Windows system, and a final one for a MacOs system.

As explained before at this chapter, the accept method is called over a router object, with a RouterVisitor as a parameter. So, inside this method, the visit method of the RouterVisitor parameter is called with the router object as input. This is the double dispatch, where at first step the router object is identified at run time as a concrete class (SpeedMaster, AlwaysOn or SwitchNet) and its accept method is called. Then, at second step, the correct instance for the RouterVisitor parameter is determined at run time and its visit method is finally called.

Therefore, thanks to the double dispatch mechanism, it is possible to find out at run time the correct instance type and to call the correct method without using if, switch/case, instanceof or similar programming constructions.

As an example, we will show the custom code of the RouterMacConfigurator class:

```
class RouterMacConfigurator implements RouterVisitor {
    public void visit(SpeedMaster sm) {
        sm.configuration = 15643;
        sm.id = “ip?10.1.1.4&mask?255.255.0.0”;
    }

    public void visit(AlwaysOn ao) {
          ao.client = 34;
          ao.configuration = “10.1.1.4:255.255.0.0”;
    }

    public void visit(SwitchNet sn) {
          sn.user = “default;255.255.0.0;10.1.1.4;”
          sn.password = “=B34&3@5”;
          sn.channel = 34;
    }
}
```

Example 11.7: Visitor class implementation example

To configure a router with the router configurator classes only have to call the visit method of the selected router configurator class implemented (at this example, RouterMacConfigurator) with the router driver instance as a parameter.

On the other hand, if our aim is to configure all the router, we can create a new method as follows at Example 11.8. This implementation implies that all drivers will be initialized.

```
public void configureAll (List rdrivers, RouterVisitor rv) {
    for (Iterator it = routerdrivers.iterator(); it.hasNext();) {
          Router rdriver = (Router) it.next()
          rdriver.accept(rv);
    }
}
```

Example 11.8: Method to apply visitor to every visitable instance

To summarize, the visitor pattern allows us to separate the object hierarchy and the methods that work over it. The explanation shown at this chapter should be a great help to understand the next chapter, and particularly the SableCC and SableCC-Ext visitor structure.

DEVELOPMENT INFRASTRUCTURE 12

12 DEVELOPMENT INFRASTRUCTURE

12.1 INTRODUCTION

At this chapter we will study some tools that have been essential in the development of our OCL 2.0 expressions processor.

Since we chose at chapter 10 the OCL 2.0 Parser Subsystem, as part of the Dresden OCL2 Toolkit [DOTw], as our preferred alternative, to understand its basic structure is mandatory. Therefore, at this stage of the development, SableCC ([SCCw] and [Kon05]) has an increasing responsibility because the major part of the OCL 2.0 Parser Subsystem development is based on that tool.

In addition, we introduce a complete explanation (a full one can be found at [Kon05]) about the OCL 2.0 Parser Subsystem and its construction process. SableCC-Ext, as an extension of SableCC, is the compilers compiler used there. To study its syntax and grammar possibilities is a precondition to Chapter 13, where the OCL 2.0 grammar of our processor is described in detail according to SableCC-Ext syntax.

To conclude this chapter, a brief description of the Integrated Development Environments used on our project is showed in order to note that programming task can become easier than we expect.

12.2 SABLECC

12.2.1 INTRODUCTION

SableCC is an object-oriented framework that generates compilers in the Java programming language. This tool allows us to obtain a compiler with a shorter development cycle, and its structure is based on an object-oriented hierarchy. As an open source project, we can study its code and adapt it in the way that we prefer.

The generated framework by SableCC includes an intuitive strictly typed abstract syntax tree and tree walker classes. With these structures, SableCC keeps a clean separation between generated code and user-written code which makes the code easier to read and maintain.

SableCC is known as compilers compiler because it is a compiler of specification files, which have a token declaration zone and a production rules zone, i.e., a grammar recognizing a specific language. The result of the compilation of this specification file is a compiler structure. When we compile such structure, we have the binary source of the compiler that we wanted to construct.

12.2.2 STEPS TO BUILD A COMPILER

SableCC specifies a concrete list of steps to follow in order to obtain a compiler through its framework. This list is shown below:

1. Create a SableCC specification file containing the token definitions and the grammar of the language that we want to recognize.
2. Execute SableCC on such specification file to obtain the generated classes of the framework.

3. Implement a tree walker class inherited from classes generated at previous step. These classes will do the treatment over the parse tree in order to convert it to another data structure or to interpret the input data from the tree to a specific output.
4. Program a main compiler class that uses both previous classes, generated and written ones.
5. Compile the main compiler class with a Java compiler in order to obtain the binary classes that form the compiler.

Another view of this list can be found at Figure 12.1.

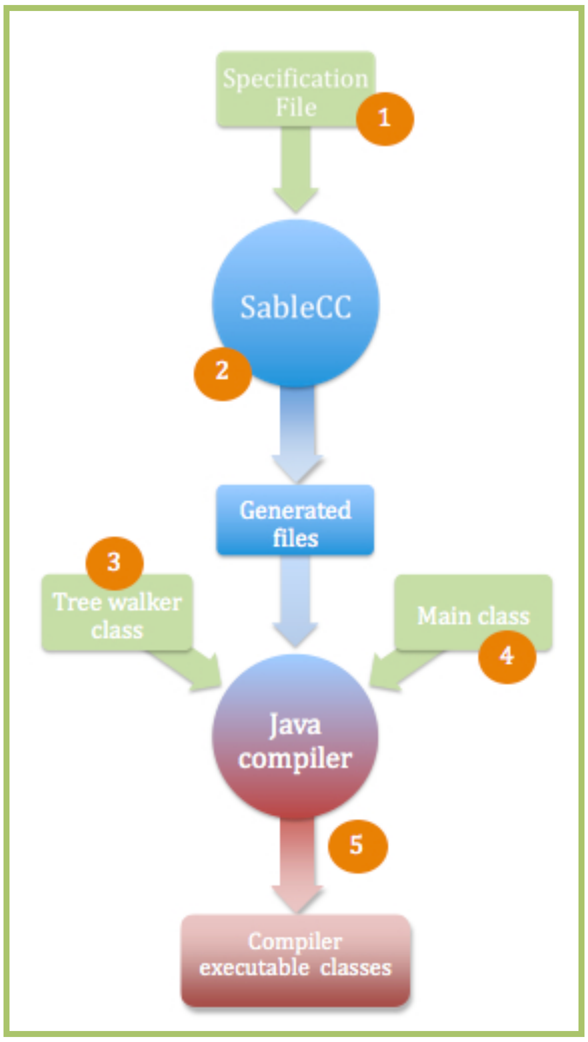


Figure 12.1: Steps to make a compiler through SableCC

12.2.3 Specification File

Once the steps to create a compiler using SableCC are known, it is necessary to focus on the specification file, which is the keystone of the SableCC-based code generation.

Such file contains different zones where to define the different elements and rules that conform our language to compile. These zones on the specification file need to be written following a concrete syntax as shown in Example 12.1.

As we can see at such example, there are five main sections in the specification file of SableCC. These sections are **`Package`**, **`Helpers`**, **`Tokens`**, **`Ignored Tokens`** and **`Productions`**. In order to understand the next part of this chapter we recommend to remember concepts shown at chapter 8, where the compilation process is described in detail.

Every specific zone will be explained, including his special syntax, during the next sections of this chapter in order to introduce the different characteristics that a well-formed specification file must fulfill.

It could be said that we can write comments in the specification file, using the double slash '//' preceding our comment line or encapsulating a more-than-one-line comment between /* and */ like in most of programming languages.

```
Package OCLcompiler;
Helpers
    alpha     =    [[‘a’ .. ‘z’]+[‘A’ .. ‘Z’]];
    decimal   =    [‘0’ .. ‘9’];
    alphadec  =    [alpha + decimal]
Tokens
    context   =    ‘context’;
    id        =    alpha alphadec*;
    inv       =    ‘inv’;
    colon     =    ‘:’;
    greqthan  =    ‘>=’;
    number    =    [‘0’ .. ‘9’]+;
    blank     =    (‘ ‘ | 13 | 10)+;
Ignored Tokens
    blank;
Productions   /* grammar */
    constraint     = context [class]:id inv [name]:id colon
                       expression;
    name           = id;
    expression     = {basic} operand
                       | {comparison} expression operation
                          operand;
    operation      = greqthan;
    operand        = {attribute} id
                       | {constant} number;
```

Example 12.1: Specification file for SableCC

12.2.3.1 Package declaration

As we saw at Example 12.1, the specification file for SableCC can begin with a package declaration. The declaration is made using the keyword **`Package`** followed by a pathname that indicates the folder where SableCC puts the generated files.

Case 1: single pathname ⇒ Package OCLcompiler;

Generated code inside package OCLcompiler

Case 2: multiple pathname ⇒ Package ocl.code.parser;

Generated code inside package parser, which is placed inside package code, which is also placed inside package ocl

Case 3: without package declaration

Generated code inside default (empty) package

Definition 12.1: Package declaration

This declaration is optional, and if it does not appear, the generated classes will be placed at the same directory where SableCC is executed, i.e., the default folder. Since SableCC uses Java programming language, if we use a specific package directory, every generated source class will use it as package in his Java file.

12.2.3.2 Helpers

At helpers zone we can describe constructions like macros, which simplify the aim writing complex tokens by means of character sets or regular expressions.

When SableCC finds a helper inside a token declaration, it replaces the helper by its declaration in a semantic way (see Example 12.2 below).

```
Helpers
  myhelper = ‘x’|’y’
Tokens
  mytoken  = ‘w’ helper ‘z’
```

The language that represents ‘mytoken’ is {‘wxz’, ‘wyz’} ***but not*** {‘wx’, ‘yz’}***.***
It is used the semantic meaning.

Example 12.2: Helper declaration

Every helper have to be written in an equality equation which left side is the helper name and the right one can be a regular expression or a character set, as we said earlier.

12.2.3.3 Tokens

As we defined earlier at chapter 8, tokens are the minimum unit that represents information. They are essential at the specification file, because are the basis of the grammar rules.

Below the label **`Tokens`**, the syntax for a token list is very simple. Only have to write an identifier followed by an equal sign and a regular expression or character set, similarly as we did with helpers.

A character set consists of a single character or a complex construction as we can found at Definition 12.2.

· ***a range:*** [‘a’ .. ‘z’] ***indicates the characters from ‘a’ to ‘z’***

· ***an union of character sets:*** [[‘a’..’z’]+[‘A’..’Z’]] ***indicates all characters***

· ***a difference of character sets:*** [[‘a’..’z’]-[‘n’+’o’]] ***indicates from ‘a’ to ‘m’ and from ‘p’ to ‘z’***

Definition 12.2: Complex character sets

On the other side, the syntax for regular expressions is very similar than syntax explained at Definition 8.6, found at chapter 8. However, there exist some little changes with SableCC implementation. SableCC supports union (using symbol ‘|’ between two expressions), concatenation (writing one expression preceding next one) and Kleene closure (using * or + symbols, according to Kleene star or Kleene plus), as well as parentheses.

Furthermore, SableCC allows to use the symbol '?' that can be used to indicate that an expression is not mandatory, i.e., it can appear or not. This optional behaviour is expressed putting the '?' symbol at the end of the expression (e.g., "exp1 exp2?" indicates that the second expression is optional and it could be omitted inside the full expression).

12.2.3.4 Ignored Tokens

At this section appear the token names that have to be ignored by the Lexer algorithm during the lexical analysis (e.g., blanks, returns, comments, tabs), which output is the input tokenized as seen at Example 8.11. Obviously, these tokens will not appear at such output.

```
Tokens
    blank   =   (' ' | 13 | 10)+;   //Kleene star of space and
                                    //return chars
    tab     =   9;            // Unicode char for tabulation
Ignored Tokens
    blank, tab;          //token names separated by commas
```

Example 12.3: Ignored tokens

12.2.3.5 Productions

The principal difference between SableCC and other compiler compilers is that SableCC does not allow using explicit program code inside the grammar production rules. As we explain earlier, one of the principles of SableCC is to achieve a perfect separation between generated and programmed code. Since that, to write custom code inside the production rules is not a possibility.

SableCC does not completely support EBNF syntax (we will describe this concept at chapter 9) for production rules, because of SableCC's design. There exist some naming rules to follow that EBNF syntax does not carry out. These naming rules are explained at Definition 12.3.

· For every production with more than one alternative, each one must be preceded with a name between curly brackets as follows:

production = {myalt1} **alt1 |** {myalt2} **alt2;**

As an exception for this rule, if there are only two alternatives, on of them can be written without name.

· Inside an alternative, if an element appears twice, it must be preceded with a name between simple brackets and connected to the element itself with a colon as in the next example:

production = [myelem1]:elem **token1** [myelem2]:elem **token2;**

The same exception explained for first case is valid for this context.

Definition 12.3: Naming alternatives and elements inside a production rule

SableCC uses the names that we chose to follow the naming rules to name its generated classes, as we will see at next sections. Therefore, to choose the most suitable names helps us when we have to program the custom code of the walker classes.

Finally, the syntax of the productions rules is very similar than that one explained at chapter 8, section 8.2.4. Below the `Productions` directive, every production rule must be written indicating the name of the production itself, a equal sign and a list of alternatives separated by '|'. Note that every production rule must finish with a semicolon.

When writing productions we have to think in what we want and how it can be expressed. To understand all the allowed constructions for the expressions of the language that we want to process is mandatory. Therefore, the easier we write the grammar, the easier it will be to understand and maintain for other people. It is highly recommendable to use comments in order to provide descriptions that can be useful even for the one that is writing.

```
Productions   /* grammar */
    constraint     = context [class]:id inv [name]:id colon
                       expression;
    name           = id;
    expression     = {basic} operand
                     | {comparison} expression operation
                         operand;
    operation      = greqthan;
    operand        = {attribute} id
                     | {constant} number;
```

Example 12.4: Productions extracted from Example 12.1

12.2.4 Generated Infrastructure

Once we have written the specification file, we have to execute SableCC with it in order to generate the classes that will conform our compiler framework.

As a compiler of language specification files, SableCC could find errors on our specification. Therefore, if it happens, we have to solve them before to continue with the compilation process.

At this subsection we will explain what we obtain through the SableCC execution by means of generating the framework of the specification file that contains the tokens, helpers and productions shown at Example 12.1.

12.2.4.1 Generation process

First of all we have to save the specification in a file (e.g., named oclgrammar) and open a terminal in our computer. Then, we should have a version of SableCC (file sablecc.jar) that can be found at [SCCw]. In the terminal we have to write and execute the command that we can see at Example 12.5.

In our terminal execute this (oclgrammar is the name of the specification file):

```
~/myfolder$  java -jar sablecc.jar oclgrammar
```

and if no error found, we obtain the next output:

```
 -- Generating parser for oclgrammar in /myfolder
Verifying identifiers.
Generating token classes.
Generating production classes.
Generating alternative classes.
Generating analysis classes.
Generating utility classes.
Generating the lexer.
 State: INITIAL
 - Constructing NFA.
......................
 - Constructing DFA.
............................................
....................
 - resolving ACCEPT states.
Generating the parser.
..............
..............
..............
..
..............
```

Example 12.5: Execution of SableCC with the specification file of Example 12.1

At the output found at the previous example we can see the different phases that SableCC needs to generate the framework from the specification file. First of all, it verifies the file and then generates the classes for every part of the specification, i.e., tokens, productions and alternatives, and tree walker classes. Finally, it creates a non-deterministic automaton in order to generate the Parser class. Such automaton is processed and converted into a deterministic one (DFA) to avoid multiple paths. Finally, the accept states are resolved and the SableCC process ends.

The output of this process follows the next folder hierarchy:

```
OCLcompiler (main folder indicated with the Package directive)
+--analysis
    -- Analysis.java
    -- AnalysisAdapter.java
    -- DepthFirstAdapter.java
    -- ReversedDepthFirstAdapter.java
+--lexer
    -- lexer.dat
    -- Lexer.java
    -- LexerException.java
+--parser
    -- parser.dat
    -- Parser.java
    -- ParserException.java
    -- State.java
    -- TokenIndex.java
+--node
    -- all node hierarchy classes ...
```

Example 12.6: Folder structure generated by SableCC

The lexer package contains the Lexer class that performs the lexical analysis. In case of error, the LexerException is thrown. On the other hand, the parser package contains the Parser class that performs the syntax analysis. In case of error, the ParserException is thrown. Also in this package we found the TokenIndex class, which maintains an identification index for every token of the specification file. At this two packages, the .dat files contains internal information about the tokens and the grammar specified before. Without them, the parser cannot work.

The analysis package contains the tree walker classes. Specifically, such package contains an interface, called Analysis.java and three walker classes that implement it.

Finally, the node package contains a class for every token, production and alternative of the specification file. At Example 12.7 we can see the classes inside this package.

```
Token.java                     Productions
-- TBlank.java                 -- PConstraint.java
-- TColon.java                 -- PExpression.java
-- TContext.java               -- PName.java
-- TGreqthan.java              -- POperand.java
-- TId.java                    -- POperation.java
-- TInv.java
-- TNumber.java                Alternatives
-- EOF.java                    -- AAttributeOperand.java
                               -- ABasicExpression.java
                               -- AComparisonExpression.java
Superclass of Productions,     -- AConstantOperand.java
alternatives and Token         -- AConstraint.java
-- Node.java                   -- AName.java
                               -- AOperation.java
```

Example 12.7: Generated classes inside node package

As we explain earlier, the names chosen to follow the naming rules of SableCC are used to identify every generated class.

The main class at package node is **`Node.java,`** which is the ancestor of all of the other classes at such package. At the same time, every class representing a token is named preceding a 'T' before the token name with the first character in upper case (e.g., token classes for colon, id or context are named TColon.java, TId.java and TContext.java, respectively).

In the case of productions, each one is represented by means of an abstract class, which name begins with a 'P' character followed by the production name with the first character in upper case (e.g., productions constraint and name are represented by PConstraint.java and PName.java, respectively).

The alternative classes extend these abstract classes. Each alternative class is named similarly than the other classes explained before. In this case, the name is formed with an 'A' character followed by both the alternative element name (if exists) and the name of the production where it belong, each one with the first character in upper case.

Each alternative class has an attribute for every element inside the alternative. These attributes are private and must be accessed using the setter and getter methods included at such class. As we explained before, if an element appears more than one time at the same alternative, it must be named according the naming rules of SableCC. This element name is now the name of the attribute inside the alternative class. The setter and getter methods are named using "set" and "get" keywords followed by the custom name of the element inside the alternative (if exists), or the current one (if not).

At next example we study the case of the context production rule seen at Example 12.1.

· Production at specification file:

constraint = context [class]:id inv [name]:id colon expression;

· Abstract class representing this production:

PConstraint.java

· Concrete class that extends this production class and represents the alternative "context [class]:id inv [name]:id colon expression;" ***:***

AConstraint.java that has the next private attributes:

```
+ private TContext _context_;
+ private TId _classifier_;
+ private TInv _inv_;
+ private TId _name_;
+ private TColon _colon_;
+ private PExpression _expression_;
```

Such class also contains the next setter and getter methods

```
+ setContext(TContext node)
+ setClassifier(TId node)
+ setInv(TInv node)
+ setName(TId node)
+ setColon(TColon node)
+ setExpressions(PExpression node)
+ getContext()
+ getClassifier()
+ getInv()
+ getName()
+ getColon()
+ getExpressions()
```

Example 12.8: Generation of constraint production of Example 12.1

12.2.4.2 Visitor Pattern adapted to SableCC

At chapter 11 we saw how the visitor pattern works. Now, we are going to explain the visitor pattern used by SableCC and implemented in its tree walker classes.

A definition of the visitor pattern could be as *"a solution to the problem of adding operations on the elements of an object structure without changing the classes of the elements on which it operates"*[GH95].

The first change of the visitor pattern implemented by SableCC with respect to the original design is the name of the main methods that realises the pattern. In SableCC (and in SableCC-Ext, as we will see later), the accept method is named apply, and each visit method of the Visitor interface is named case method. According to the SableCC authors, these names are more appropriated for the visitor pattern concept.

The case methods are named adding the name of the element that it belongs after the case word. As an example, for the token Colon (TColon class, as we explain earlier) the case method name is caseTColon, with a TColon parameter.

Moreover, the Visitor interface is named Switch, and the Node class, which is the ancestor of all AST classes, implements an interface called Switchable, similar than the Visitable one. Such Switchable interface contains the description of the apply (remember, accept) method in order to be implemented for every class that implements this interface, i.e., all the classes inherited from Node.

According to this explanation, the SableCC structure for the visitor pattern looks like at Figure 12.2.

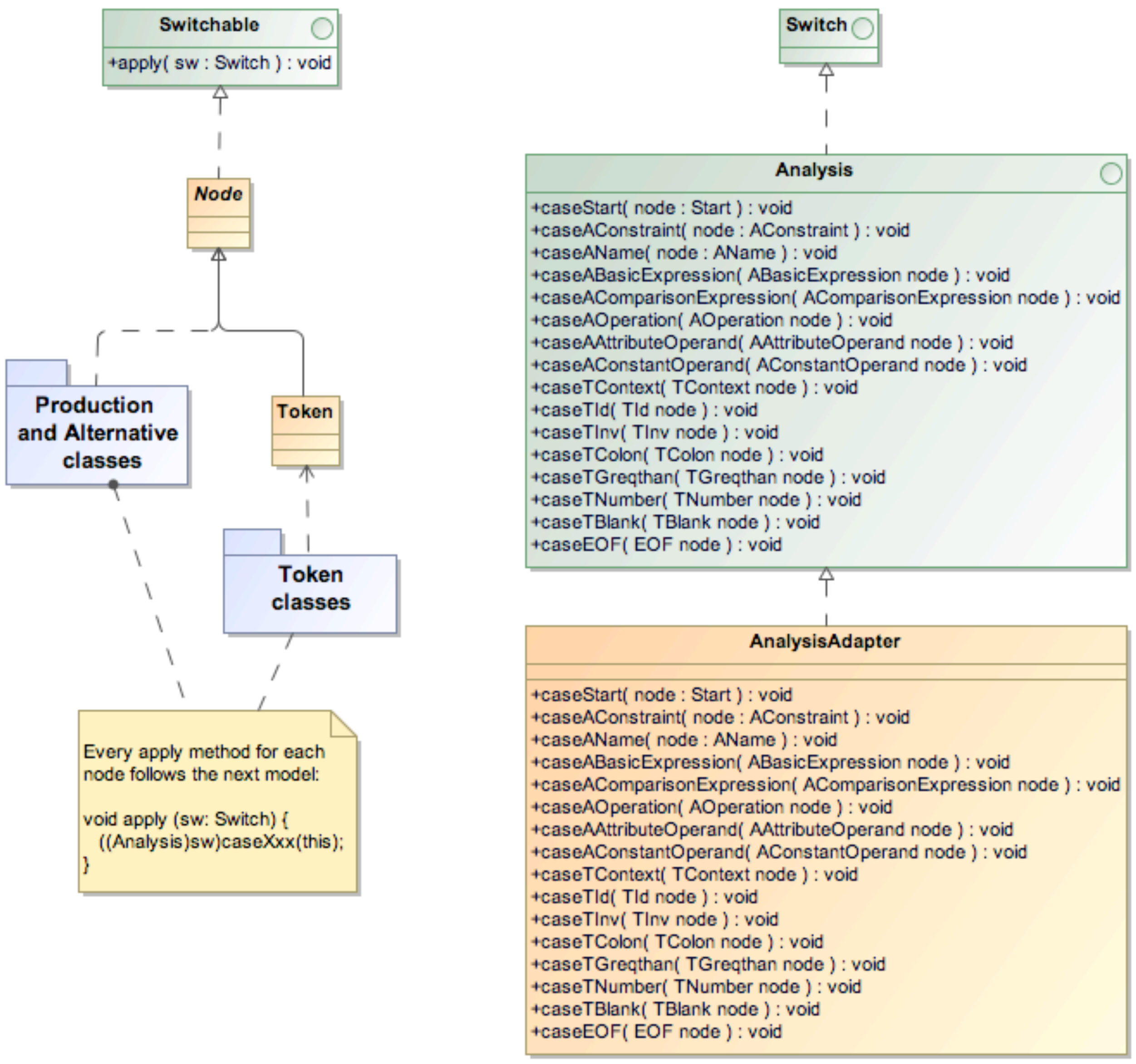

Switchable
+apply(sw : Switch) : void
Switch
Node
Production and Alternative classes
Token
Token classes
Every apply method for each node follows the next model:
void apply (sw: Switch) {
((Analysis)sw)caseXxx(this);
}
Analysis
+caseStart(node : Start) : void
+caseAConstraint(node : AConstraint) : void
+caseAName(node : AName) : void
+caseABasicExpression(ABasicExpression node) : void
+caseAComparisonExpression(AComparisonExpression node) : void
+caseAOperation(AOperation node) : void
+caseAAttributeOperand(AAttributeOperand node) : void
+caseAConstantOperand(AConstantOperand node) : void
+caseTContext(TContext node) : void
+caseTId(TId node) : void
+caseTInv(TInv node) : void
+caseTColon(TColon node) : void
+caseTGreqthan(TGreqthan node) : void
+caseTNumber(TNumber node) : void
+caseTBlank(TBlank node) : void
+caseEOF(EOF node) : void
AnalysisAdapter
+caseStart(node : Start) : void
+caseAConstraint(node : AConstraint) : void
+caseAName(node : AName) : void
+caseABasicExpression(ABasicExpression node) : void
+caseAComparisonExpression(AComparisonExpression node) : void
+caseAOperation(AOperation node) : void
+caseAAttributeOperand(AAttributeOperand node) : void
+caseAConstantOperand(AConstantOperand node) : void
+caseTContext(TContext node) : void
+caseTId(TId node) : void
+caseTInv(TInv node) : void
+caseTColon(TColon node) : void
+caseTGreqthan(TGreqthan node) : void
+caseTNumber(TNumber node) : void
+caseTBlank(TBlank node) : void
+caseEOF(EOF node) : void

Figure 12.2: SableCC visitor structure generated for Example 12.1

We can see that Switch interface is empty and Switchable one has only the apply method. The introduction of these low-restrictive interfaces allows us to add new elements to the visit engine without modifying any existing class.

Therefore, to add a new element, we define a new interface implementing Switch interface, with a new case method for every new element. The apply method for such elements will call the case methods of this new interface. So, the previous object structure remains equal than before.

The Analysis interface denoted at the last figure includes the headers for every case method to implement. As a utility class, SableCC provides us with the AnalysisAdapter class that contains a default implementation for each case method and implements the Analysis interface.

12.2.4.3 AST Walker classes

SableCC provides two AST walker classes that extend AnalysisAdapter class. First one, called DepthFirstAdapter visits the nodes in a normal depth-first traversal (see Example 12.9), and second one, called ReversedDepthFirstAdapter, makes the visit in a reverse depth-first traversal (see Example 12.10).

```
class DepthFirstAdapter extends AnalysisAdapter
{
    public void caseAConstraint(AConstraint node)
    {
        inAConstraint(node);
        node.getContext().apply(this);
        node.getClassifier().apply(this);
        node.getInv().apply(this);
        node.getName().apply(this);
        node.getColon().apply(this);
        node.getExpression().apply(this);
        outAConstraint(node);
     }
}
```

Example 12.9: Depth-first traversal for constraint alternative case method

It is necessary to note that these two default tree walker classes provide two methods for each case method of a node class that allows the programmer to introduce custom code inside this predefined traversal. These methods are called inXxx and outXxx, where the Xxx is the name of the case alternative or token.

As we can check at both Example 12.9 and Example 12.10, these methods are called before and after visiting a node. By the way, at these examples how to traverse the hierarchical structure of nodes is shown by means of calling the apply method of each attribute. Such attributes are the descendant nodes obtained thanks to the respectively getter method that belongs to the parent node being visited, as we explained earlier.

```
class ReversedDepthFirstAdapter extends AnalysisAdapter
{
    public void caseAConstraint(AConstraint node)
    {
        inAConstraint(node);
        node.getExpression().apply(this);
        node.getColon().apply(this);
        node.getName().apply(this);
        node.getInv().apply(this);
        node.getClassifier().apply(this);
        node.getContext().apply(this);
        outAConstraint(node);
     }
}
```

Example 12.10: Reverse depth-first traversal for constraint alternative case method

To conclude, if we want to create a custom tree walker class, we only have to develop a new class extending one of the two default classes shown before (DepthFirstAdapter or ReversedDepthFirstAdapter) overriding the necessary inXxx and outXxx method.

Nevertheless, we also can use these classes as a template and implement our custom tree walker class by our own. But this alternative has an increased level of difficulty and is error-prone.

12.2.4.4 Compiler compilation and usage

Once we have implemented our preferred tree walker class, the next step is to program a compiler class with the main method to execute. This task is very simple due to the structure of the framework generated by SableCC according to our specification file.

The code for this Compiler class should be similar than the next Java example.

```java
class Compiler
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Type a constraint:");

        // Creation of the Lexer
        InputStreamReader in = new InputStreamReader(System.in);
        PushBackReader pbr = new PushBackReader(in, 1024);
        Lexer lexer = new Lexer(pbr);

        // Creation of the parser
        Parser parser = new Parser (lexer);

        // Parse the input constraint
        Start parsetree = parser.parse();

        // Start the visitor pattern analysis
        parsetree.apply(new MyDepthFirstAdapter());
        // MyDepthFirstAdapter is our tree walker class extending
        // the DepthFirstAdapter class
    }
}
```

Example 12.11: Compiler class

As we can see at Example 12.11, we only have to create the lexer with the preferred input (at this example, the input is the standard input of the terminal where we execute the Compiler class). The next step is to create the parser instance with the lexer as a parameter, and to parse the input.

Finally, we have to start the process made by the tree walker class that will do some specific actions with the parse tree (e.g., create the AST from the parse tree, or translate the input into another language). This action is very simple. We only have to init the visitor process by means of calling the apply method of the Start node class, which always is our initial node because it is the output of the parse method. Once we have program the Compiler class, we only have to compile it and to execute it to check if all it's ok.

To summarize, at this section we have understood how SableCC can help us to develop a compiler following all the required steps, from the creation of a specification file to the programming of a main Compiler class.

12.3 OCL2.0 PARSER SUBSYSTEM

12.3.1 INTRODUCTION

The OCL 2.0 Parser Subsystem is the OCL 2.0 expressions processor of the Dresden OCL2 Toolkit. This tool has a similar functionality than our processor. But these tools also have a set of differences. If interesting, a complete explanation of this subsystem can be found at [Kon05].

Nevertheless, the structure used by the Dresden team is a perfect base in which begin our development. Throughout this phase, the OCL2.0 Parser Subsystem infrastructure and source code will become in our user manual for implement our processor.

Due to this design decision, to study and explain what is and how this tool works is mandatory. Then, firstly we will show the structure and the construction process of the OCL 2.0 Parser Subsystem along this chapter. And at next chapter we will see the changes needed in SableCC to adapt it to OCL 2.0 particularities.

12.3.2 Structure

The OCL 2.0 Parser Subsystem has four main modules. Three of them are the known compiler phases studied before at chapter 8, that is, a lexical and syntax analyzers (lexer and parser, respectively) and an attribute evaluator, which can be seen as a semantic analyzer.

The fourth module is an AST node factory that creates the Abstract Syntax Tree nodes on demand. This factory is based on a hash map storing (node name, method that creates the node) pairs.

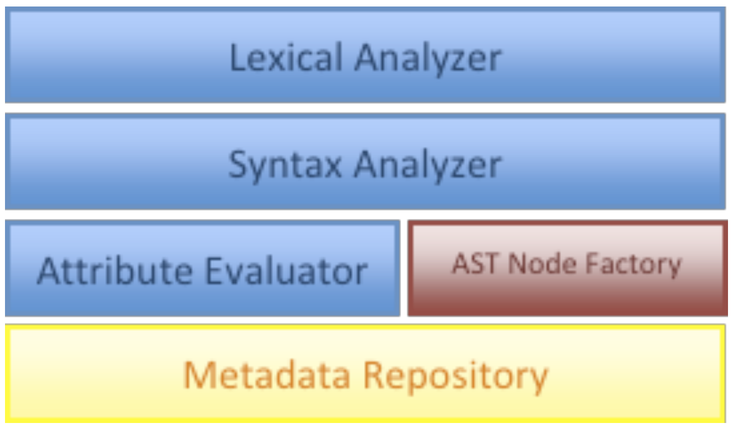


Figure 12.3: OCL 2.0 Parser Subsystem modules

This processor of OCL 2.0 expressions is connected with a UML model stored in a metadata repository. The attribute evaluator makes a match between the UML elements used in the OCL 2.0 input constraints and their representation in the repository (if exists) in order to find errors and return feedback to users.

As we saw at earlier chapters, the lexical analyzer divides and organizes the input into tokens. Then, the syntax analyzer transforms the token stream into a concrete syntax tree according to the specification file. And finally, the attribute evaluator converts the CST into the abstract syntax tree according to the OCL 2.0 metamodel.

12.3.3 Development of the OCL2.0 Parser Subsystem

At this point of the chapter, to explain how to construct the OCL 2.0 Parser Subsystem is mandatory to know what are the steps to change or adapt with a view to the development of our OCL 2.0 expressions processor.

Firstly, an extended version of SableCC, that is, a SableCC-Ext version is needed. At section 12.4 we will show how to obtain it. Once we have the SableCC-Ext version, the next step is to write a specification file like we did earlier generating a compiler with SableCC. Nevertheless, this specification file has some changes in comparison with SableCC specifications. All these changes will be shown at section 12.4.3.

Then, executing SableCC-Ext with our specification file containing the tokens, and productions defining OCL 2.0 will result in the generation of an OCL 2.0 lexer and an OCL 2.0 parser. These two files have the responsibility of create the parse tree, also known as concrete syntax tree (CST).

In this generation process we also obtain an attribute evaluator base class, which is essential in the new visitor structure defined by SableCC-Ext's design. Such attribute evaluator is an abstract class that implements the SwitchWithReturn interface, which is similar than Switch interface as was seen before. All this changes in the visitor pattern will be explained in detail in following sections.

The base class must be extended implementing the required methods with the necessary rules to convert the CST nodes to their AST form as indicated in the OCL 2.0 specification.

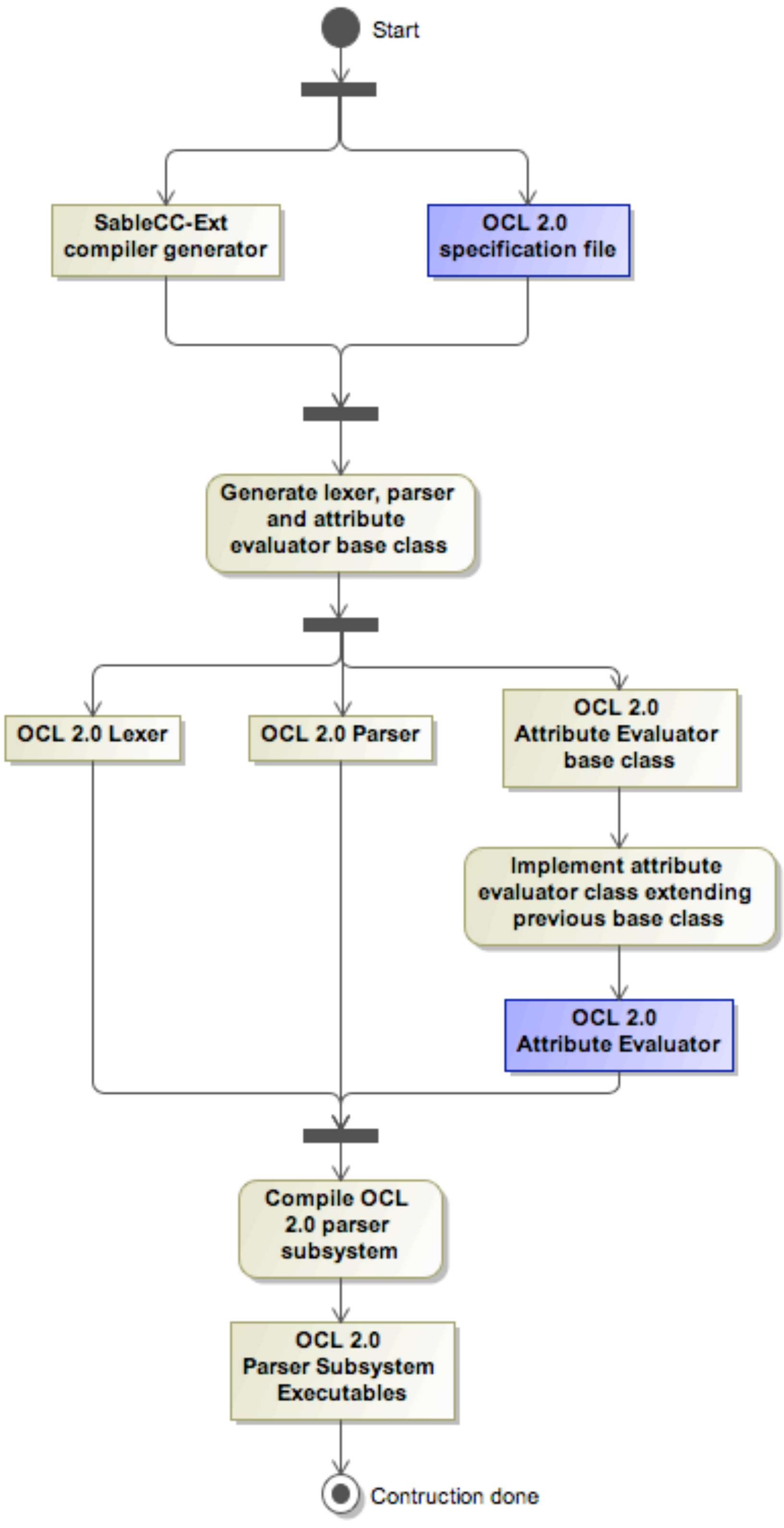

Start
SableCC-Ext compiler generator
OCL 2.0 specification file
Generate lexer, parser and attribute evaluator base class
OCL 2.0 Lexer
OCL 2.0 Parser
OCL 2.0 Attribute Evaluator base class
Implement attribute evaluator class extending previous base class
OCL 2.0 Attribute Evaluator
Compile OCL 2.0 parser subsystem
OCL 2.0 Parser Subsystem Executables
Contruction done

Figure 12.4: OCL 2.0 Parser Subsystem Construction

A full node evaluator will be derived joining the two classes. The base class is responsible to make a correct node traversal, and our extension constructs the node conversion.

After that, we have to create the main Compiler class mixing the three previous components, i.e., the OCL 2.0 lexer and parser and the attribute evaluator, in the same way that we did before with SableCC.

Final step consists in to compile this Compiler Java class with all its dependencies and to execute it in order to check the correct behaviour of the OCL 2.0 Parser Subsystem.

At Figure 12.4 we can see in blue colour the modules that we have to develop by hand. On the other side, the rest of shapes indicate processes or modules made or generated automatically.

12.4 SABLECC-EXT

12.4.1 INTRODUCTION

At this section we introduce SableCC-Ext as the compiler generator chosen to construct the framework for our OCL 2.0 expressions processor. The reason for this decision is because SableCC-Ext is the generator of the OCL 2.0 Parser Subsystem of the Dresden OCL2 Toolkit, and has all the functionality required by our project.

At next parts of this chapter we will study SableCC-Ext's creation, the changes that it implies in both the specification file and the node visitor process in order to make easier to understand the following chapters that explain our OCL 2.0 grammar and the development process, among other things.

12.4.2 Why extend SableCC

As we explained before, SableCC is a good tool to generate the necessary framework to develop a compiler. Nevertheless, there exist some languages to compile that need special features that SableCC is not able to provide. In this set of more complex languages we found the OCL 2.0.

The main limitation of SableCC generated framework is found at the tree walker classes. They are unusable for the attribute evaluator of the OCL 2.0 Parser Subsystem.

Such tree walker classes provide the methods inXxx and outXxx that are only called before and after the descent into the child nodes of a current node. But OCL 2.0 needs to inject custom code between the descent to each child node.

The reason for this required behaviour is the data dependencies that exist in the evaluation of the OCL 2.0 nodes. Since OCL is a left-to-right language, there exist data needed by the siblings of a node to compute their own evaluation.

Moreover, these data dependencies should be resolved by passing the required information to the nodes that could need it, but SableCC not allows introducing additional parameters in the visitor infrastructure to descent into other nodes. Another possibility is to return this information by each node when its evaluation finish, but SableCC visitor pattern design implies that both the apply and caseXxx methods are void.

To understand the data dependencies, look at next OCL 2.0 constraint:

```
context Person::salary : Real init: self.agesWorking*1000.00
```

Analysing this simple constraint, we can see that exists information to share between the differente sections of such constraint.

Dividing the constraint in parts, we found:

- `Person::salary`, ***the constrained element***
- `Real`, ***the type of the constrained element***
- `self.agesWorking*1000.00`, ***the initializer expression that is made by:***
    - `self.agesWorking`, ***a property to consult, that belogns to self***
    - `1000.00`, ***a real constant***

A list of data dependencies should be:

- `Person::salary` ***must return its type to be compared with the type written in the constraint (in this case*** `Real`***).***
- `self.agesWorking` ***needs to know the type of*** `self` ***in order to check if*** `agesWorking` ***exists in such type. This type must be passed when*** `Person::salary` ***is computed.***
- `self.agesWorking` ***has to return its type in order to check if it matches with*** `1000.00` ***type.***
- `self.agesWorking*1000.00` ***has to return its operation result type in order to check if it matches with*** `Person::salary` ***type.***

As we have seen, even a simple constraint hides a lot of dependencies.

Example 12.12: Constraint dependencies

The Dresden team, in the development of their OCL 2.0 Parser Subsystem as a part of the Dresden OCL2 Toolkit, found these handicaps and solved them by extending the basic version of SableCC with additional features that provided the needed behaviour without losing the power of the original.

12.4.3 SableCC-Ext Changes

The SableCC-Ext generation process was done using SableCC. The Dresden team wrote a grammar specifying the changes that SableCC-Ext must fulfill. Such grammar, as a specification file, was the input of a new SableCC execution. On the other hand, the source code of SableCC was adapted to produce as output for every execution the abstract class that is the attribute evaluator skeleton.

Then, the output of this process was a complete version of the SableCC extension, known as SableCC-Ext that provides the necessary functionalities for the OCL 2.0 language.

As we explained before, this new version allows passing information between a node and its descendants and vice versa. The top-down sharing is made by passing a parameter with the necessary information to the descendants in the apply method. As well, the bottom-up sharing is made by a return parameter in each apply method, which represents the AST node processed for each descendant.

With this design, a node can share information with its descendants introducing a parameter (defined as inherited information) into the header call of the apply method, and then each method will return a parameter with the AST information (defined as synthesized information) that can be essential for the current node.

To customize the generation of the attribute evaluator base class, SableCC-Ext introduces four directives, that is, #chain, #nocreate, #customheritage and #maketree, which change the behaviour of such skeleton. These directives can be used in the specification file for SableCC-Ext and will be explained at section 12.4.5.

Such file has a new syntax that must be known. Every production rule must indicate what is the type of the AST node that returns once the evaluation process finishes. This type name must appear between by the production name and the equal sign and must be flanked by angle brackets. By the way, this type name usage can be applied to alternatives in a production.

Other change is the possibility to mark a token or alternative element with a '!' symbol preceding such token or alternative. This action involves that the token or alternative marked will be skipped during the attribute evaluation. This way, the syntactic sugar[1] can be erased.

These ones are part of SableCC-Ext changes with respect to common SableCC. The main change is the new visitor structure that will be introduced at next section.

12.4.4 New Extended Visitor Architecture

The new visitor architecture included in SableCC-Ext can be found at Figure 12.5. It is important to note that previous SableCC visitor classes are still generated, but are not shown at image in order to improve understandability.

Now, Switch and Switchable interfaces are SwitchWithReturn and SwitchableWithReturn, which pay attention on the parameter return functionality added.

At the same way than with SableCC, SwitchableWithReturn defines the apply method that includes a new object parameter as input in addition to the SwitchWithReturn one. It also returns another object that will be the AST node object computed for each node.

On the other hand, SwitchWithReturn interface is empty and it is implemented by AnalysisWithReturn interface that contains the same case methods than in Analysis class, but with a new parameter as input.

[1] syntactic sugar: additions to the syntax of a computer language that do not affect its functionality but make it "sweeter" for humans to use.

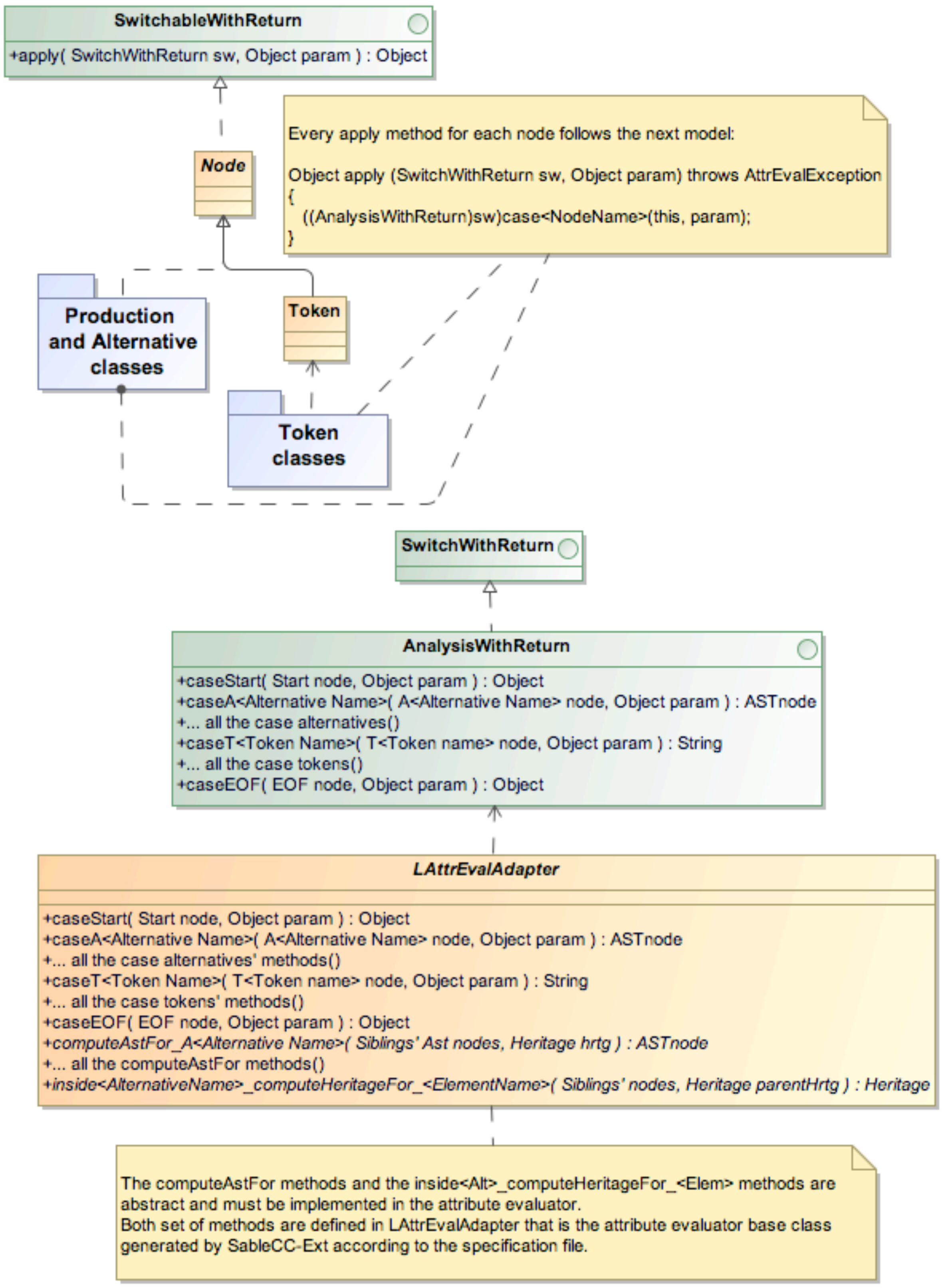

SwitchableWithReturn
+apply(SwitchWithReturn sw, Object param) : Object
Every apply method for each node follows the next model:
Object apply (SwitchWithReturn sw, Object param) throws AttrEvalException
{
((AnalysisWithReturn)sw)case<NodeName>(this, param);
}
Node
Production and Alternative classes
Token
Token classes
SwitchWithReturn
AnalysisWithReturn
+caseStart(Start node, Object param) : Object
+caseA<Alternative Name>(A<Alternative Name> node, Object param) : ASTnode
+... all the case alternatives()
+caseT<Token Name>(T<Token name> node, Object param) : String
+... all the case tokens()
+caseEOF(EOF node, Object param) : Object
LAttrEvalAdapter
+caseStart(Start node, Object param) : Object
+caseA<Alternative Name>(A<Alternative Name> node, Object param) : ASTnode
+... all the case alternatives' methods()
+caseT<Token Name>(T<Token name> node, Object param) : String
+... all the case tokens' methods()
+caseEOF(EOF node, Object param) : Object
+computeAstFor_A<Alternative Name>(Siblings' Ast nodes, Heritage hrtg) : ASTnode
+... all the computeAstFor methods()
+inside<AlternativeName>_computeHeritageFor_<ElementName>(Siblings' nodes, Heritage parentHrtg) : Heritage
The computeAstFor methods and the inside<Alt>_computeHeritageFor_<Elem> methods are abstract and must be implemented in the attribute evaluator.
Both set of methods are defined in LAttrEvalAdapter that is the attribute evaluator base class generated by SableCC-Ext according to the specification file.

Figure 12.5: SableCC-Ext Visitor Pattern Structure

These case methods also contain the return AST node, as synthesized attribute to share in bottom-up direction. Each method is implemented in LAttrEvalAdapter abstract class.

LAttrEvalAdapter also contains the headers for the computeAstFor methods, which are the methods that make the attribute evaluation (semantic analysis) and create the corresponding AST node for every node class. These methods contain as parameters the previous siblings' AST nodes and a special data structure called Heritage.

The Heritage contains all needed information about the context of the node (e.g., what is the contextual class or the constrained element in the constraint in process, what is the class of self keyword, what iterator variables can be used inside the body expression of an iterator, what is the source expression of an operation, and so on).

The Heritage can be computed by means of the inside<Alternative Name>_computeHeritageFor_<Element Name>, which is processed before to descent into an element node of an alternative node. The name of this method is made adding the alternative node name and the element name inside the places denoted with angle brackets in the previous definition. As we will se next to it, this method is not mandatory. If needed, we must use the directive #customheritage at the end of an alternative in the specification file, to create it in the skeleton class. If this directive does not appear, a copy of the parent node heritage is passed instead to the descendants.

Then, since we have a skeleton class that makes the tree traversal and call the all nodes for itself, as specified by the specification file, our job is to extend this class to compute the computeAstFor and the custom heritage methods in order to create the AST nodes and make the AST tree linking them.

LAttrEvalAdapter, as a skeleton class, contains the case methods implementation that follows the next algorithm:

1. Begin with the Start node and call its apply method, with a class extending LAttrEvalAdapter and an empty Heritage as parameters.
2. The apply method calls the case method for the current node, and this method do the following actions:
   - 2.1. Obtain one of its descendant nodes to be processed,
   - 2.2. If custom heritage object needed, call the inside_computeHeritageFor_ method for the current descendant in order to calculate its Heritage
   - 2.3. Call the apply method for the current descendant with the Heritage as an additional parameter. Get the return AST node object and store it.
   - 2.4. Repeat from 2.1 to here for every descendant that has not been processed.
3. Call the AST node factory to create the AST node of the current node being processed. This behaviour can be skipped marking the alternative with the #nocreate alternative.
4. Call the computeAstFor method for the current node with the Heritage and the AST node object of every previous descendant node processed, as parameters. This parameters contain the AST node of step 3, if exists. The return of this method is the AST of the current node.
5. Return this AST node.

Finally, it is important to note that the name of the skeleton class is not casual. LAttrEvalAdapter is the attribute evaluator base class name because the OCL 2.0 grammar is an l-attribute grammar. It means that all productions of the grammar have an attribute assigned to evaluate according to a set of rules (that can be found at [Obj06]). And the first 'l' indicates that the algorithm to make the attribute evaluation uses a left-to-right depth-first tree walk.

12.4.5 Attribute Evaluator

At this section we will show in detail the attribute evaluator inside the visitor architecture of SableCC-Ext with an example.

First of all we must complete the explanation of the specification file directives #chain, #nocreate, #customheritage and #maketree.

Directive #chain is used to note that the alternative marked with it must contain only one element inside, and its synthesized attribute have to be returned as synthesized attribute of this alternative directly. It skips to do unnecessary processing.

On the other hand, #nocreate directive delegates the responsibility to create the AST node for an alternative or element to the computeAstFor method instead of to the caseXxx method directly.

Also, the #customheritage directive indicates that the element marked with it needs a custom code to compute its Heritage object. The result of this directive is the creation of the inside..computeHeriteageFor.. method to implement.

Look at next production of an hypothetical OCL 2.0 specification file:

```
let_exp_cs <LetExp> =    let   [variables]:initialized_variable_list_cs
                         in  [expression]:expression #customheritage ;
```

Note that the AST node type of this production is LetExp, as indicated between angle brackets.

The #customheritage directive implies the generation of the insideALetExpCs_computeHeritageFor_Expression method that in this case probably create an Heritage with the let variables inside because expression needs to know them to its computation.

Example 12.13: Example of #customheritage directive

Finally, #maketree indicates that the responsibility to keep and store the left siblings' AST nodes is delegated to the computeAstFor method o the currently processed node. Therefore, such method produces an AST tree from a CST list of nodes.

At this moment, we will study how SableCC-Ext works with an example based on the If construction of the OCL 2.0 language. First of all, we must to write the necessary tokens and productions inside the specification file according to its new syntax.

```
if_exp_cs <IfExp> =    if [condition]:logical_exp_cs
                       then [then_branch]:ocl_expression_cs
                       else [else_branch]:ocl_expression_cs
                       endif  ;
logical_exp_cs <...> = ... ;
ocl_expression_cs <...> = ... ;
```

Example 12.14: Production for OCL 2.0 If construction

The first step of the visitor pattern when someone calls the apply method of an AIfExpCs node, which is the alternative name for the grammar production if_exp_cs in Example 12.14, is to call the case method of this alternative, i.e., the caseAIfExpCs method.

```
public Object apply(SwitchWithReturn sw, Object param)
                                     throws AttrEvalException
{
    return ((AnalysisWithReturn) sw).caseAIfExpCs(this, param);
}
```

Example 12.15: Apply method for If construction of Example 12.14

The Object parameter called param in Example 12.15 is the Heritage that is shared in a top-down traverse.

The case method generated automatically inside the LAttrEvalAdapter abstract class (the attribute evaluator skeleton class) should be similar than as follows:

```
public final IfExp caseAIfExpCs(AIfExpCs node, Object param)
                                  throws AttrEvalException {
   // All this code is generated automatically by SableCC-Ext
   Heritage nodeHrtg = (Heritage) param;
   Heritage childHrtg = null;
   // Descending to condition
   PLogicalExpCs childCondition = node.getCondition();
   OclExpression astCondition = null;
   if( childCondition != null) {
         astCondition = (OclExpression)
         childCondition.apply(this, nodeHrtg.copy());
   }
   // Descending to then branch
   POclExpressionCs childThenBranch = node.getThenBranch();
   OclExpression astThenBranch = null;
   if( childThenBranch != null) {
         astThenBranch = (OclExpression)
         childThenBranch.apply(this, nodeHrtg.copy());
   }
   // Descending to else branch
   POclExpressionCs childElseBranch = node.getElseBranch();
   OclExpression astElseBranch = null;
   if( childElseBranch != null) {
         astElseBranch = (OclExpression)
         childElseBranch.apply(this, nodeHrtg.copy());
   }
   // create AST node for current CST node here.
   // If #nocreate appears, the next line will be skipped
   IfExp myAst = (IfExp) factory.createNode("IfExp");
   // Next method is abstract here and must be implemented
   myAst = computeAstFor_AIfExpCs(myAst, nodeHrtg, astCondition,
                                  astThenBranch , astElseBranch);
   return myAst;
}
```

Example 12.16: Case method for if expression inside LAttrEvalAdapter skeleton generated by SableCC-Ext automatically

Finally, as we explained earlier, we have to implement a concrete class that extends LAttrEvalAdapter with both the computeAst and the custom heritage methods completed.

Therefore, SableCC-Ext provides a framework that allows us to delegate the task of make the node traversal to a generated skeleton class. Our job is only to complete the abstract methods defined at such skeleton with their correct evaluation inside another class in order to create the AST nodes properly.

Such new class must extend the abstract one. This design allows us to have different implementations in different files, and to use the one that we decide in every moment.

12.5 INTEGRATED DEVELOPMENT ENVIRONMENTS

12.5.1 INTRODUCTION

An Integrated Development Environment is a software tool that provides facilities to computer programmers for software development.

When the number of classes and code lines to manage and compile become huge, to continue with the development process is a hard job. To avoid this problem, IDEs integrate different systems such as source code editors, compilers, interpreters or debuggers that deal with our code and make our job easier.

Examples of IDEs supporting Java language are Eclipse and NetBeans.

12.5.2 ECLIPSE

Eclipse is the IDE used in the whole development phase of this project. It can be found at [ECLw] and a full manual is [Dau04].

Using Eclipse allows us to have a perfect error control of our code. It also contains syntax highlighting and code completion. Such aids were essential during the programming phase because the source code becomes evil and wild without them.

In addition, there exist a huge amount of plug-ins for Eclipse that provide more functionality to this tool. As an example, to connect with a control version repository is possible. In our case, the conceptual modeling environment (EinaGMC) tool used in the project was stored and maintain in one on-line repository, therefore to synchronize with it in order to always have the last version of the code was a very simple job through Eclipse.

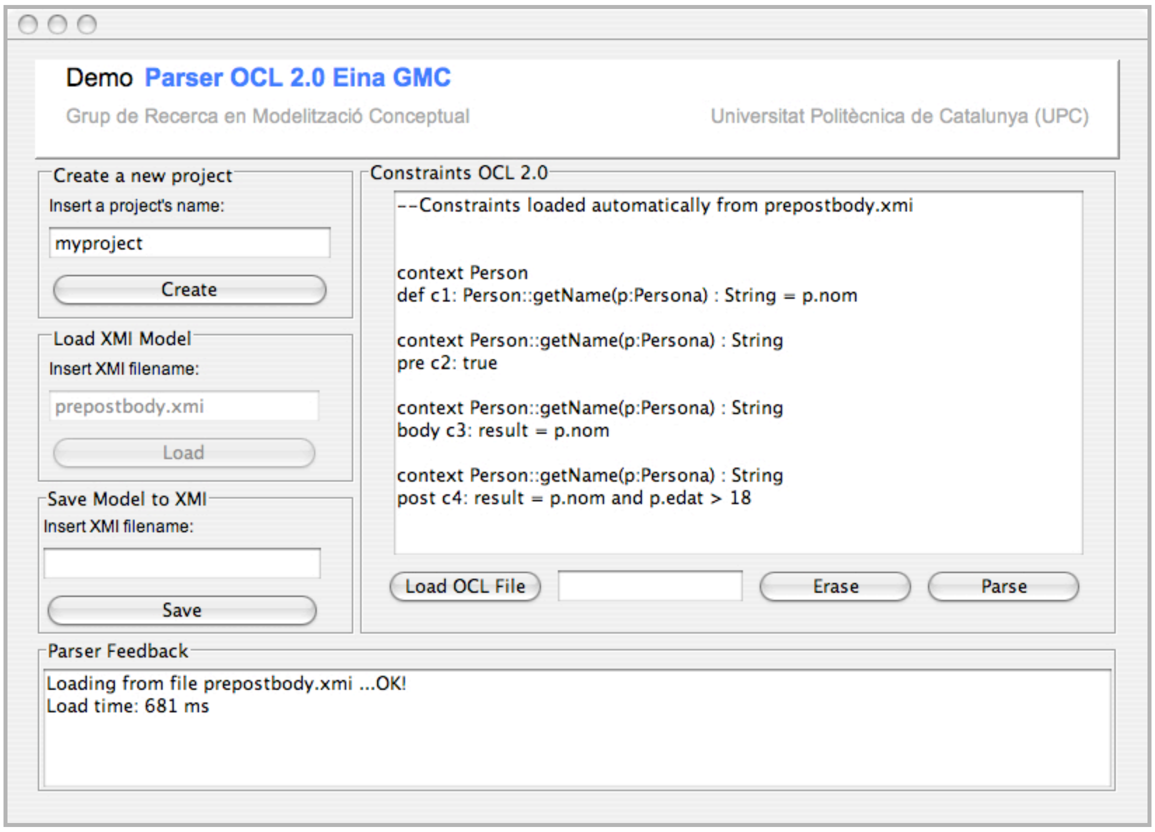
12.5.3 NETBEANS

On the other hand we have NetBeans [NETw], as another alternative IDE to use in the development process of a software system.

Nevertheless, in our case NetBeans was only used to develop the GUI (Graphical User Interface) for our Demo program that test the correctness of the OCL expressions processor. NetBeans provide an easy-to-use interface to create GUIs by means of drag and drop the elements that we want to appear in the user interface to a window containers.

With only the power of a mouse, you can construct the interface like a Lego system, i.e., selecting the visual elements and placing them in a custom position inside a container.

Once we have the interface made, NetBeans automatically generates the code that controls the buttons, text fields and so on.



```
--Constraints loaded automatically from prepostbody.xmi


context Person
def c1: Person::getName(p:Persona) : String = p.nom

context Person::getName(p:Persona) : String
pre c2: true

context Person::getName(p:Persona) : String
body c3: result = p.nom

context Person::getName(p:Persona) : String
post c4: result = p.nom and p.edat > 18
```

```
Loading from file prepostbody.xmi ...OK!
Load time: 681 ms
```

Figure 12.6: Screenshot of the Demo used to check the correct behaviour of our processor

In the development process of the Demo explained before, the code generated for this interface by NetBeans was edited and customized inside Eclipse because this IDE has a more comfortable look in its MacOS X version, and seems to be more adapted to MacOS X graphical window system.

OCL 2.0 GRAMMAR 13

13 OCL 2.0 GRAMMAR

13.1 INTRODUCTION

At previous chapters we have explained the concept of grammar as a structure that is used to recognize a concrete language. Furthermore, we showed the special syntax for writing grammars as input for SableCC and SableCC-Ext.

At this point we will introduce the OCL 2.0 grammar used as the core of our processor of OCL expressions. It is important to note that this grammar is strongly based in OCL grammar of the Parser Subsystem of the Dresden OCL2 Toolkit [DOTw], which is similar than OCL grammar explained at chapter 9 of the OCL specification document [Obj06].

A complete view of our grammar can be found at *B OCL 2.0 grammar* section at the end pages of this document. In this chapter we will explain the main sections of this grammar including a description of the important characteristics of the production rules and diagrams showing the generated classes by SableCC-Ext for each rule within package node of the generated framework.

13.2 ABOUT GRAMMARS

As we introduced in previous chapters, our OCL 2.0 grammar is a L-attribute grammar. An attribute grammar is a formal way to define attributes for the productions of a formal grammar, associating these attributes to values. The evaluation occurs in the nodes of the abstract syntax tree.

Such attributes are can be synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent nodes to its descendants. Synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down it.

This grammar is also a L-attribute grammar because attributes are evaluated in one left-to-right depth-first traversal.

13.3 OCL 2.0 GRAMMAR DESCRIPTION

To continue with our grammar description we will explain the most important production rules of the OCL 2.0 grammar in next sections.

13.3.1 CONSTRAINT STRUCTURES

As you should know there are different kinds of constraints depending on the context where they have to be applied. Next production rules show this differentiation in order to divide the processing in three branchs.

First rule allows writing more than one constraint because grammar provides parsing a list of constraints, as shown in *context_declaration_list_cs* production. Note that directive *#nocreate* is used in order to avoid calling to the factory to create the AST node for *tail* element. Remember that each node is created through a factory class.

The creation of such resultant AST for *tail* of this production rule element will be done inside the processing of this element. Therefore it is not necessary to call the factory here.

```
context_declaration_list_cs <List> =
       [context]:context_declaration_cs
       [tail]:context_declaration_list_cs? #nocreate
    ;

context_declaration_cs <OclContextDeclaration> =
      {classifier} <OclClassifierContextDecl>
       context [context_name]:path_name_cs
       [constraints]:classifier_constraint_cs+ #customheritage
    | {attr_or_assoc} <OclAttrOrAssocContextDecl>
       context [context_name]:path_name_cs
       colon [type]:type_specifier
       [constraints]:init_or_der_value_cs+ #customheritage
    | {operation}   <OclOperationContextDecl>
       context [context_name]:path_name_cs
       [signature]:operation_signature_cs
       [constraints]:operation_constraint_cs+ #customheritage
    ;
```

Definition 13.1: Grammar rules for constraint structures

Furthermore, result of *context_declaration_list_cs* production is a List containing the ASTs of all constraints, as indicated between angle brackets.

Next production rule contains the alternatives that conform the three branches. An element inside such previous list of constraints can be a classifier constraint, an attribute or association constraint, or an operation constraint. It is important to emphasize here that we maintain the usage of several intermediate classes introduced by Dresden OCL2 Toolkit that provides an easy management of AST nodes within the tree traversal. We must highlight that such class names begin with the "Ocl" tag and they do not belong to the OCL metamodel. As we said, they are used to facilitate the processing of the ASTs providing a wrapper that allows passing more than one element as a parameter for a parent or descendant node.

In each alternative of this production we can see how is made each context header. Note that all begin with *context* token. At next section we will explain the particular productions of every one.

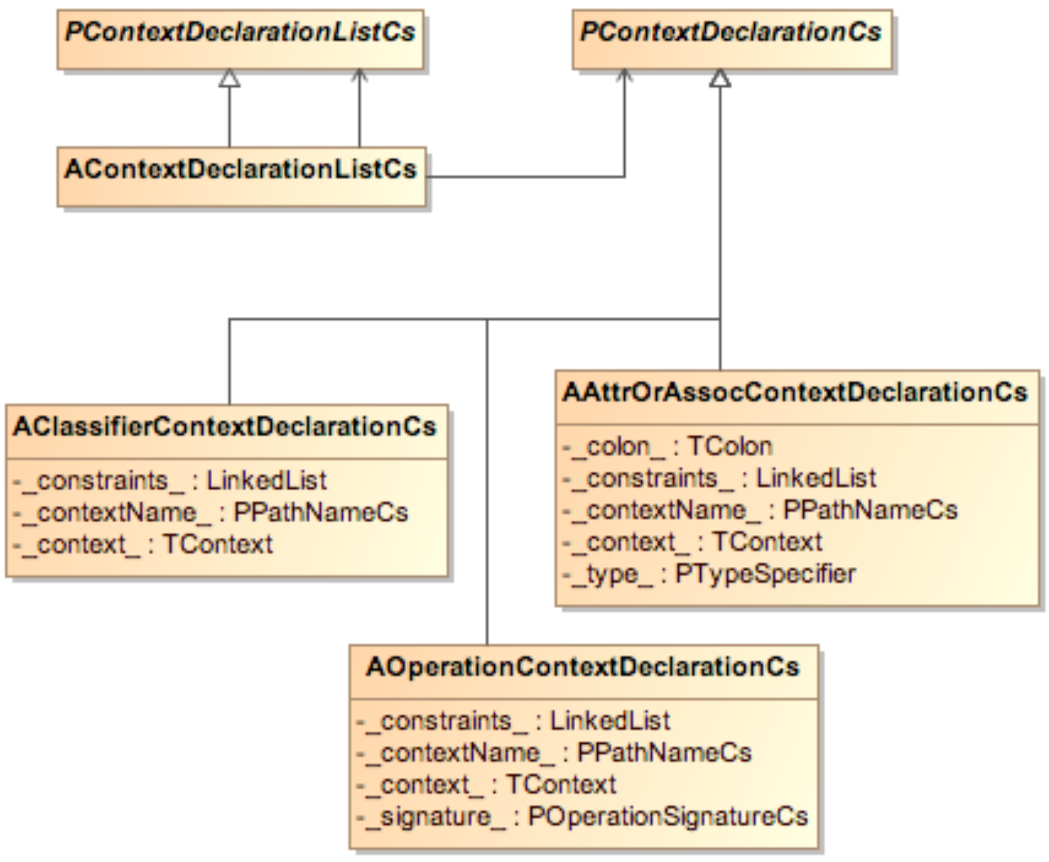

PContextDeclarationListCs
PContextDeclarationCs
AContextDeclarationListCs
AClassifierContextDeclarationCs
-_constraints_ : LinkedList
-_contextName_ : PPathNameCs
-_context_ : TContext
AAttrOrAssocContextDeclarationCs
-_colon_ : TColon
-_constraints_ : LinkedList
-_contextName_ : PPathNameCs
-_context_ : TContext
-_type_ : PTypeSpecifier
AOperationContextDeclarationCs
-_constraints_ : LinkedList
-_contextName_ : PPathNameCs
-_context_ : TContext
-_signature_ : POperationSignatureCs

Figure 13.1: SableCC-Ext generated classes for constraint structures

Figure 13.1 shows the generated classes by SableCC-Ext once we have passed the previous productions to its process. If we spend a seconds comparing such diagram with the related grammar rules we will see their relation between these two elements.

13.3.1.1 Operation signature

When we are writing a constraint for an operation we have to write its header inside the header of the constraint. To do this, we introduce the next grammar rules that describe how to construct such header.

It is very easy to understand because contains a list of parameters between parentheses and a return type at the end. Note that the name of the operation is not placed here because it was introduced at previous *context_declaration_cs* rule.

```
operation_signature_cs <OclOperationSignature> =
        paren_open
        [parameters]:formal_parameter_list_cs?
        paren_close
        [return_type]:operation_return_type_specifier_cs?
    ;

operation_return_type_specifier_cs <Classifier> =
        colon [return_type]:type_specifier #chain
    ;
```

Definition 13.2: Grammar rules for operation signatures

Another time, the relation between production rules and their generated classes by SableCC-Ext is very simple. Here only have one alternative for each production.

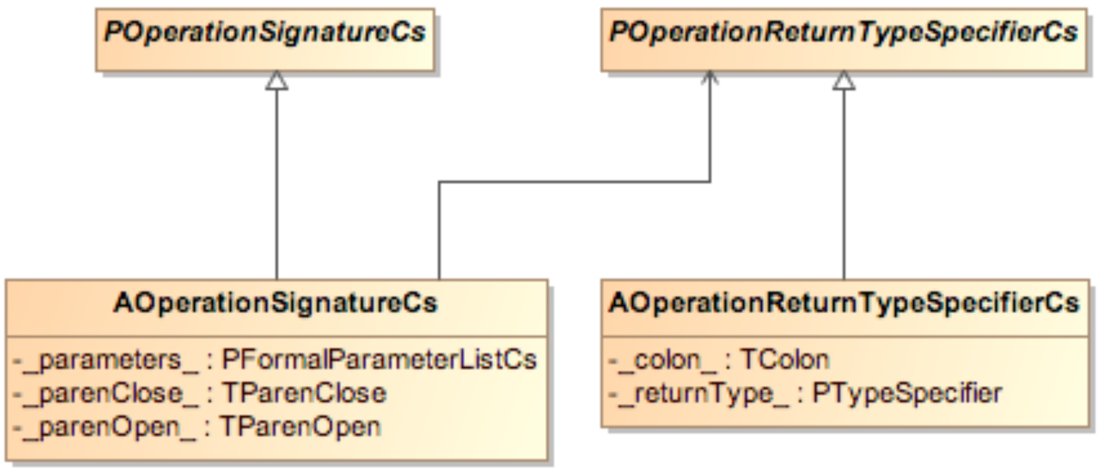

POperationSignatureCs
POperationReturnTypeSpecifierCs
AOperationSignatureCs
-_parameters_ : PFormalParameterListCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen
AOperationReturnTypeSpecifierCs
-_colon_ : TColon
-_returnType_ : PTypeSpecifier

Figure 13.2: SableCC-Ext generated classes for operation signature

It is important to note that obviously, each production is an abstract class beginning with a P in its name. Its alternatives are concrete subclasses of it with an A in the beginning of its name.

13.3.1.2 Invariants and definitions

First kinds of constraints are invariants and definitions, which share the same context construction. As we can see, invariants are only OCL expressions but definitions include an entity declaration that can be an attribute or an operation.

```
classifier_constraint_cs <OclClassifierConstraint> =
      {invariant} <OclInvariantClassifierConstraint>
        inv [name]:simple_name? colon
        [invariant]:ocl_expression_cs
    | {definition} <OclDefinitionClassifierConstraint>
        def [name]:simple_name? colon
        [definition]:definition_constraint_cs
    ;

definition_constraint_cs <OclDefinitionConstraint> =
        [entity]:defined_entity_decl_cs
        !equals
        [definition]:ocl_expression_cs #customheritage
    ;

defined_entity_decl_cs <OclDefinedEntityDecl> =
      {attribute} <OclAttributeDefinedEntityDecl>
        [attribute]:formal_parameter_cs
    | {operation} <OclOperationDefinedEntityDecl>
        [operation_name]:simple_name
        [operation]:operation_signature_cs
    ;
```

Definition 13.3: Grammar rules for invariants and definitions

It is important to note that "!" symbol before a token indicates that such token must be ignored in the parsing process.

Attribute declarations use the same production as a formal parameter of an operation signature because, as we will see when explaining parameters, both need a name and a type to be defined.

In case of operation declaration it is only necessary to reuse *operation_signature_cs* production to describe the new defined operation.

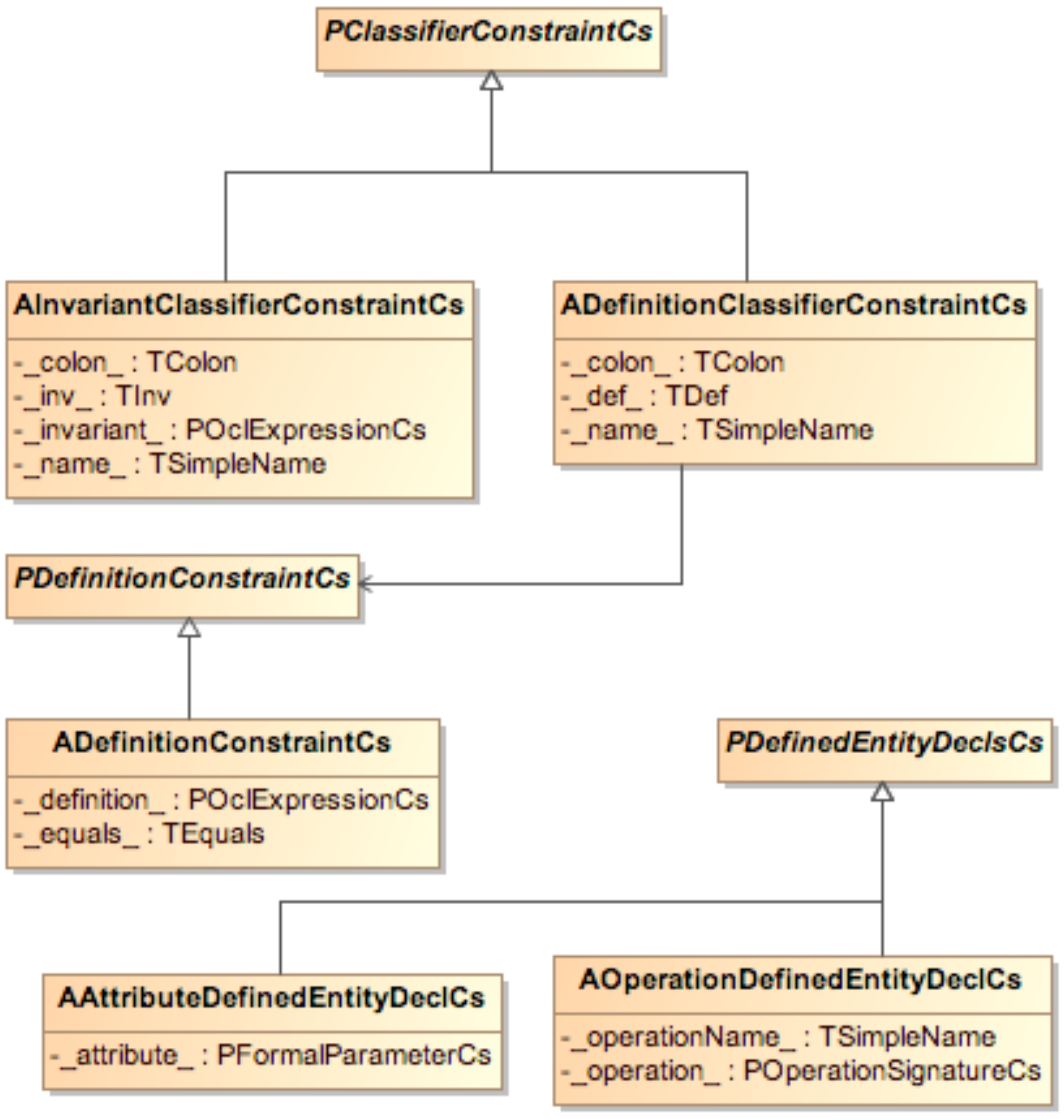

PClassifierConstraintCs
AInvariantClassifierConstraintCs
-_colon_ : TColon
-_inv_ : TInv
-_invariant_ : POclExpressionCs
-_name_ : TSimpleName
ADefinitionClassifierConstraintCs
-_colon_ : TColon
-_def_ : TDef
-_name_ : TSimpleName
PDefinitionConstraintCs
ADefinitionConstraintCs
-_definition_ : POclExpressionCs
-_equals_ : TEquals
PDefinedEntityDeclsCs
AAttributeDefinedEntityDeclCs
-_attribute_ : PFormalParameterCs
AOperationDefinedEntityDeclCs
-_operationName_ : TSimpleName
-_operation_ : POperationSignatureCs

Figure 13.3: SableCC-Ext generated classes for invariants and definitions

Figure 13.3 shows the generated classes for the invariant and definition rules of the grammar introduced before. Here we can observe the two subclasses of production *defined_entity_decls_cs*, one for attributes and the other one for operation definitions.

13.3.1.3 Initializations and derivations

Second kinds of constraints are initializations and derivations. As shown in the next piece of code the only difference between these constraint constructions is the preceding token: init or derive.

```
init_or_der_value_cs <OclAttrOrAssocConstraint> =
      {init} <OclInitConstraint>
        init [name]:simple_name? colon
        [initializer]:ocl_expression_cs
    | {derive} <OclDeriveConstraint>
        derive [name]:simple_name? colon
        [derive_expression]:ocl_expression_cs
    ;
```

Definition 13.4: Grammar rules for initializations and derivations

The generation of the classes is very obvious here due to both alternatives are very similar. It is important to emphasize that *simple_name* here is a token representing a String that starts with an alphabetic (and not numeric) character.

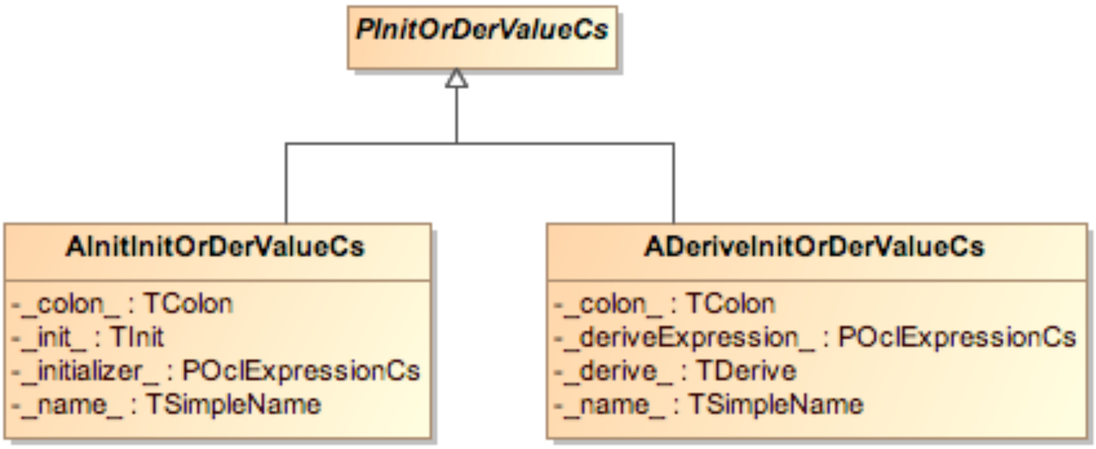


Figure 13.4: SableCC-Ext generated classes for initializations or derivations

13.3.1.4 Preconditions, postconditions and body expressions

Finally, last kinds of constraints are preconditions, postconditions and body expressions. All three have the same construction elements including the stereotype of the constraint (pre, post or body), the name of the constraint and the OCL expression of it.

```
operation_constraint_cs <OclOperationConstraint> =
        [stereotype]:op_constraint_stereotype_cs
        [name]:simple_name? colon
        [expression]:ocl_expression_cs #customheritage
    ;

op_constraint_stereotype_cs <OclOperationConstraintStereotype> =
        {pre} pre       #nocreate
    |   {post} post     #nocreate
    |   {body} body     #nocreate
    ;
```

Definition 13.5: Grammar rules for operation constraints

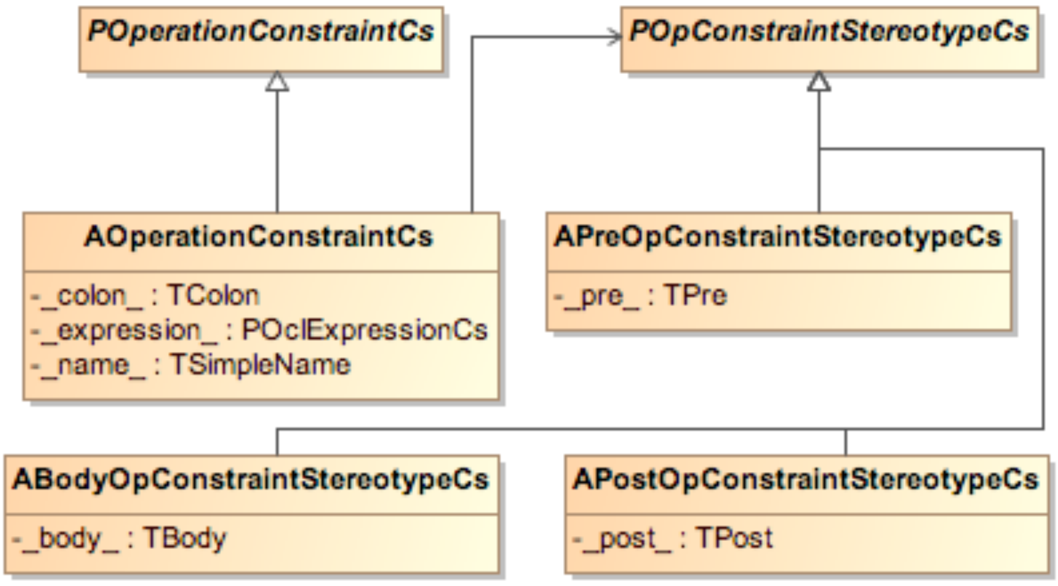

POperationConstraintCs
POpConstraintStereotypeCs
AOperationConstraintCs
-_colon_ : TColon
-_expression_ : POclExpressionCs
-_name_ : TSimpleName
APreOpConstraintStereotypeCs
-_pre_ : TPre
ABodyOpConstraintStereotypeCs
-_body_ : TBody
APostOpConstraintStereotypeCs
-_post_ : TPost

Figure 13.5: SableCC-Ext generated classes for operation constraints

13.3.2 NAMES AND IDENTIFIERS

In our grammar, identifiers can be simple String names, the iterate token, an iterator name (e.g., forAll, exists, one, ...) or an operation name. On the other hand, a path name is a concatenation of identifiers with a double colon "::" between them.

```
identifier_cs <String> =
      {simple} simple_name #chain
    | {iterate} iterate #chain
    | {iterator_name} iterator_name_cs #chain
    | {ocl_op_name} ocl_op_name #chain
    ;

path_name_cs <List>   =
    [qualifier]:path_name_head_cs* [name]:identifier_cs #nocreate;

path_name_head_cs <String> =
    identifier_cs dbl_colon #chain;
```

Definition 13.6: Grammar rules for names and identifiers

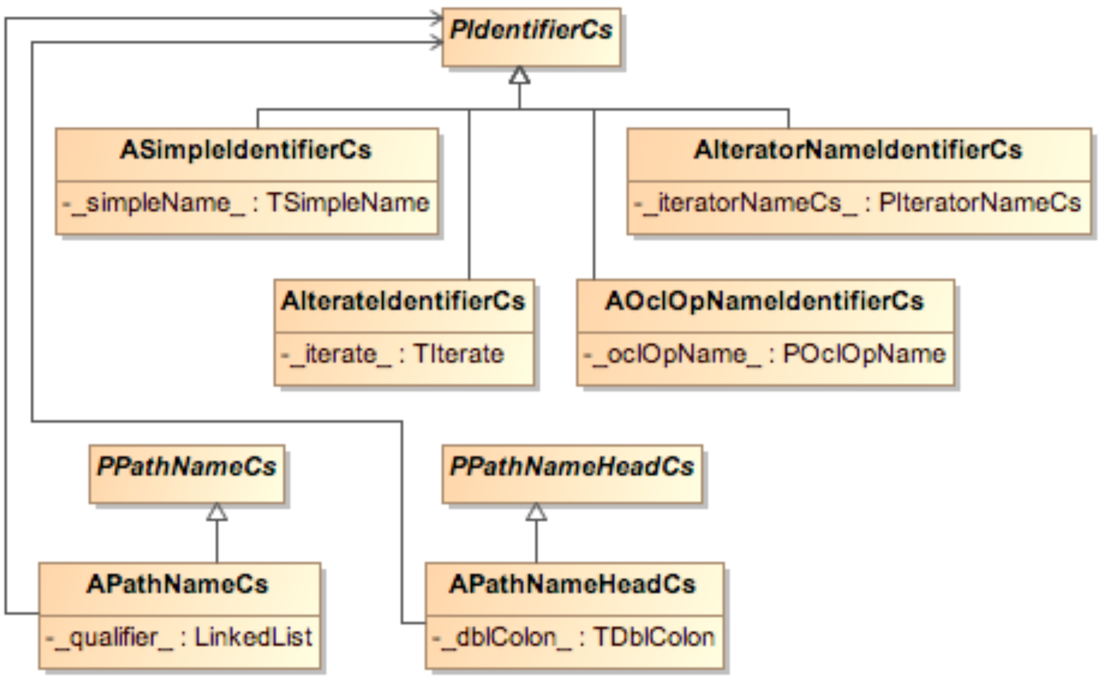


Figure 13.6: SableCC-Ext generated classes for names and identifiers

13.3.3 OCL Expressions

In this section we will describe the grammar rules for to construct and recognize OCL expressions.

13.3.3.1 Principal expressions

In our grammar we separate OCL expressions in two different branches, expressions with let or without let. An OCL expression with let is a let expression where one or more than one variables can be initialized to be used within the expression of the let construction.

```
ocl_expression_cs <OclExpression> =
      {with_let}    let_exp_cs      #chain
    | {without_let} logical_exp_cs  #chain
    ;

let_exp_cs <LetExp> =
        let [variables]:initialized_variable_list_cs
        in  [expression]:expression #customheritage #nocreate
    ;

if_exp_cs <IfExp> =
     if [condition]:logical_exp_cs
        then [then_branch]:ocl_expression_cs
        else [else_branch]:ocl_expression_cs
     endif
    ;

expression <OclExpression> = ocl_expression_cs #chain;
```

Definition 13.7: Grammar rules for principal expressions

As we will see in next sections an OCL expression without let is an expression that could be a logical expression. We use logical expression here due to the OCL 2.0 operators' precedence order. Logical operators have a lower precedence importance so logical expressions must be the first in the expression's hierarchy.

Note that both branches have the *#chain* directive in their rules. It indicates that the synthesized attribute of an OCL expression is calculated in its descendants and here it's not necessary to do any processing.

When we have a rule like A->B->C->D, we can use #chain directive to directly transfer AST node computed within D to C, and then from C to B, and finally from B to A without extra processing.

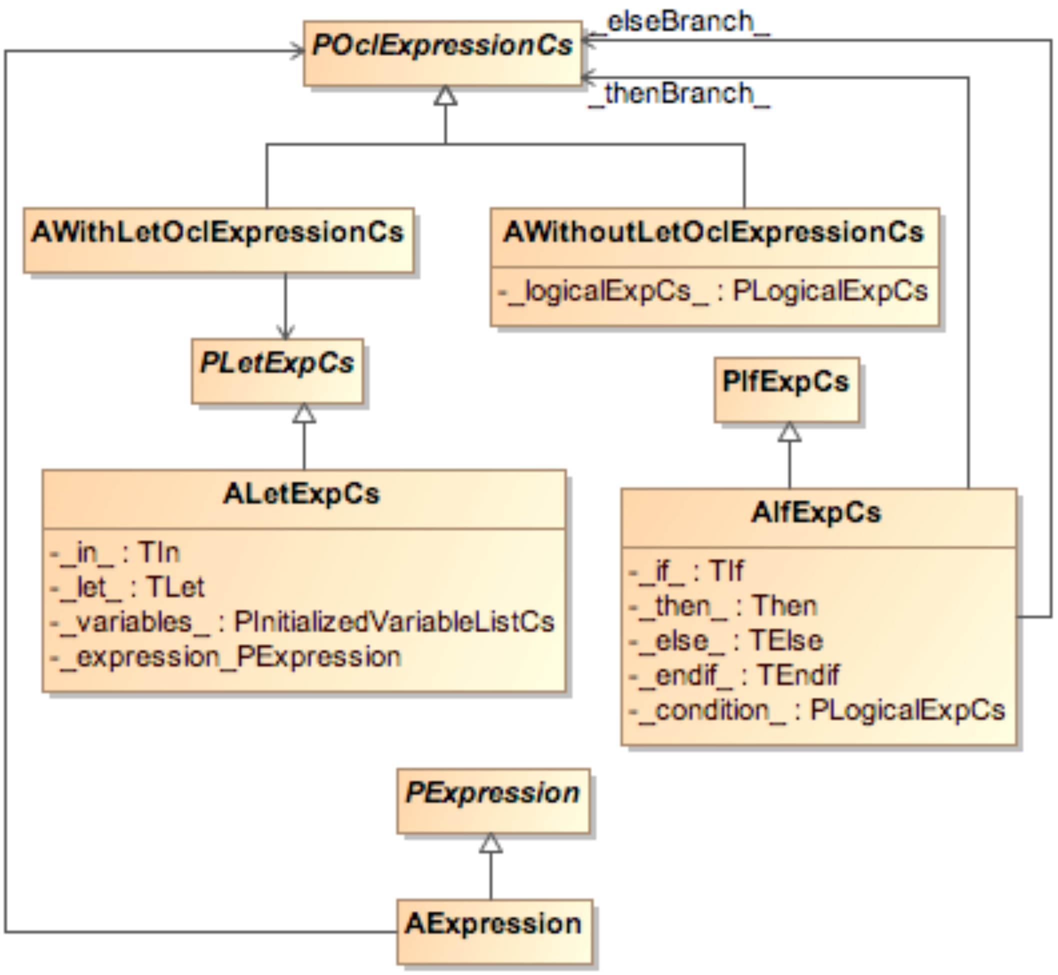


Figure 13.7: SableCC-Ext generated classes for principal expressions

In the diagram we can observe that attributes beginning with a T in their name are Token objects.

13.3.3.2 Logical expressions

A logical expression can be the chained result of a relational expression, or a binary operation between a relational expression and other logical expression containing an operator and a relational expression. Logical operators allowed here are *and*, *or*, *xor* and *implies* operators.

```
logical_exp_cs <OclExpression> =
      {chain} <OclExpression> [operand]:relational_exp_cs #chain
    | {binary} <OperationCallExp> [operand]:relational_exp_cs
         [tail]:logical_exp_tail_cs+
    ;

logical_exp_tail_cs <OclBinaryExpTail> =
      [operator]:logic_op [operand]:relational_exp_cs
    ;
```

Definition 13.8: Grammar rules for logical expressions

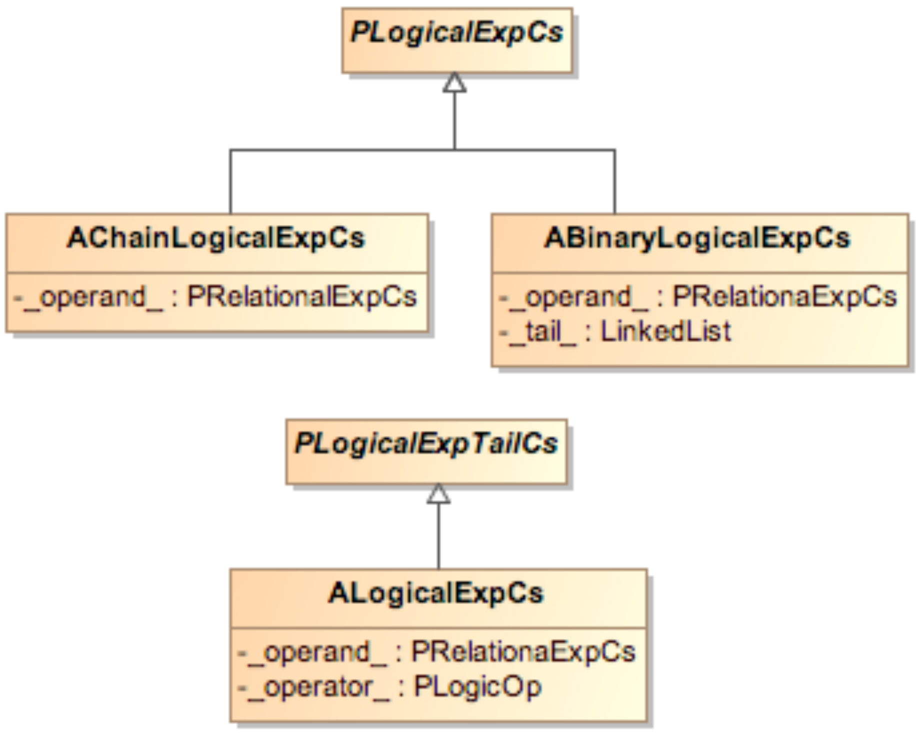


Figure 13.8: SableCC-Ext generated classes for logical expressions

13.3.3.3 Relational expressions

A relational expression can be the chained result of an additive expression, or a binary operation between an additive expression and other relational expression containing an operator and an additive expression. Relational operators allowed here are =, <>, <=, >=, <, and > operators.

```
relational_exp_cs <OclExpression> =
      {chain} [operand]:additive_exp_cs #chain
    | {binary} <OperationCallExp> [operand]:additive_exp_cs
            [tail]:relational_exp_tail_cs
    ;

relational_exp_tail_cs <OclBinaryExpTail> =
        [operator]:rel_op [operand]:additive_exp_cs
    ;
```

Definition 13.9: Grammar rules for relational expressions

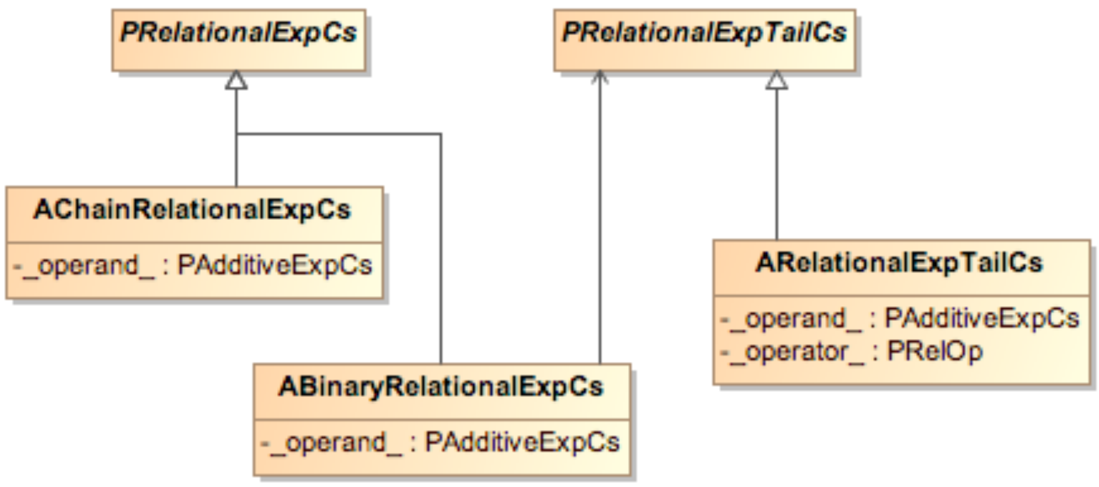


Figure 13.9: SableCC-Ext generated classes for relational expressions

It is important to emphasize that here we place productions as attributes of alternative classes instead of use a relationship between such two parts due to the aim of improve the understandability of the diagrams.

13.3.3.4 Additive expressions

An additive expression can be the chained result of a multiplicative expression, or a binary operation between a multiplicative expression and other additive expression containing an operator and a multiplicative expression. Relational operators allowed here are + and – operators.

```
additive_exp_cs <OclExpression> =
      {chain} [operand]:multiplicative_exp_cs #chain
    | {binary} <OperationCallExp> [operand]:multiplicative_exp_cs
          [tail]:additive_exp_tail_cs+
    ;

additive_exp_tail_cs <OclBinaryExpTail> =
        [operator]:add_op [operand]:multiplicative_exp_cs
    ;
```

Definition 13.10: Grammar rules for additive expressions

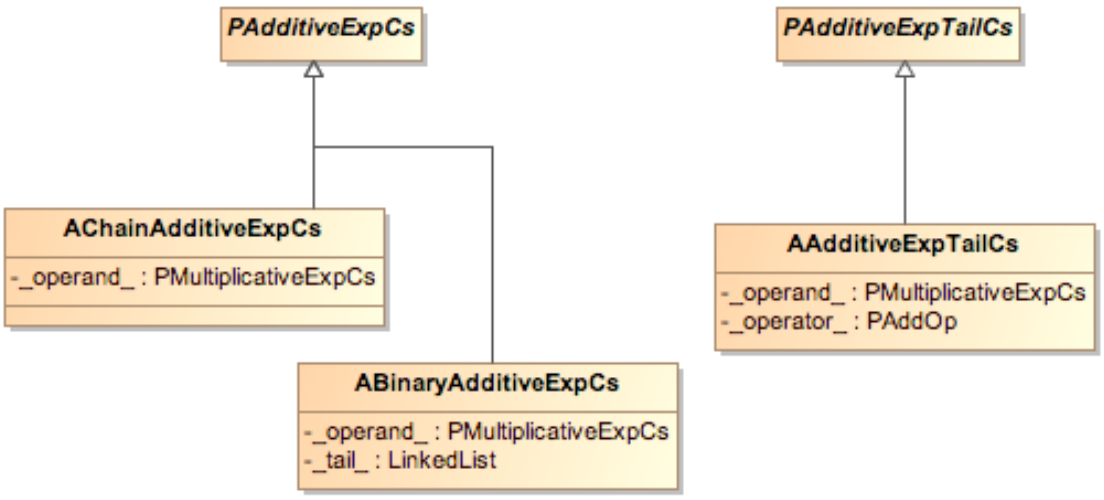


Figure 13.10: SableCC-Ext generated classes for additive expressions

13.3.3.5 Multiplicative expressions

A multiplicative expression can be the chained result of a unary expression, or a binary operation between a unary expression and other multiplicative expression containing an operator and a unary expression. Relational operators allowed here are ∗ and / operators.

```
multiplicative_exp_cs <OclExpression> =
      {chain} [operand]:unary_exp_cs #chain
    | {binary} <OperationCallExp> [operand]:unary_exp_cs
          [tail]:multiplicative_exp_tail_cs+
    ;

multiplicative_exp_tail_cs <OclBinaryExpTail> =
        [operator]:mult_op [operand]:unary_exp_cs
    ;
```

Definition 13.11: Grammar rules for multiplicative expressions

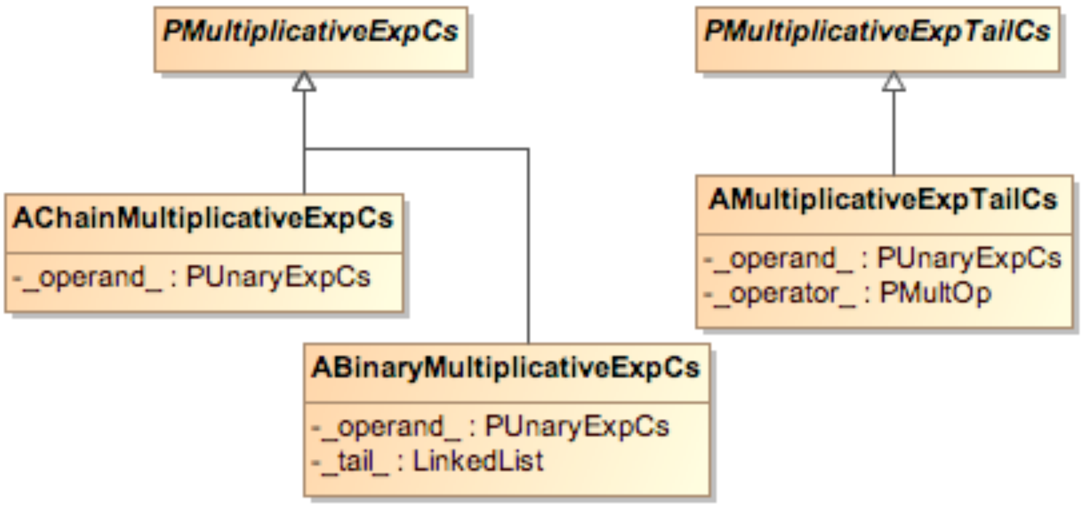


Figure 13.11: SableCC-Ext generated classes for multiplicative expressions

13.3.3.6 Unary expressions

Last elements in the hierarchy of expressions are the unary expressions. Such expressions can be a unary operator followed by a postfix expression, or the chained result of a postfix expression. Unary operations are *not* and unary minus "–".

First alternative results in an OperationCallExp as shown in the next definition.

A postfix expression can be a primary expression or a primary expression followed with a postfix tail expression. Primary expressions are literals, expressions between parentheses, property calls or if expressions.

```
unary_exp_cs <OclExpression> =
      {unary_op} <OperationCallExp> [operator]:unary_op
          [operand]:postfix_exp_cs
    | {unary_nop}   [postfix]:postfix_exp_cs  #chain
    ;

postfix_exp_cs <OclExpression> =
      {primary}[primary]:primary_exp_cs #chain
    | {with_tail}   [leftmost_exp]:primary_exp_cs
          postfix_exp_tail_cs+ #maketree #nocreate
    ;

postfix_exp_tail_cs <OclExpression> =
      {prop}  dot [prop_call]:property_call_exp_cs #customheritage
          #nocreate
    | {arrow_prop} arrow_right [tail]:arrow_property_call_exp_cs
          #chain
    ;

primary_exp_cs <OclExpression> =
      {literal} literal_exp_cs #chain
    | {parenthesized} paren_open expression paren_close #chain
    | {property} [prop_call]:property_call_exp_cs #customheritage
          #nocreate
    | {if} if_exp_cs #chain
    ;
```

Definition 13.12: Grammar rules for unary expressions

On the other hand a postfix tail expression is a property call expression preceded by a dot token or an arrow property call expression preceded by an arrow token. Such constructions are the right part of a call expression like *expression1.expression2* or *expression1->expression2*.

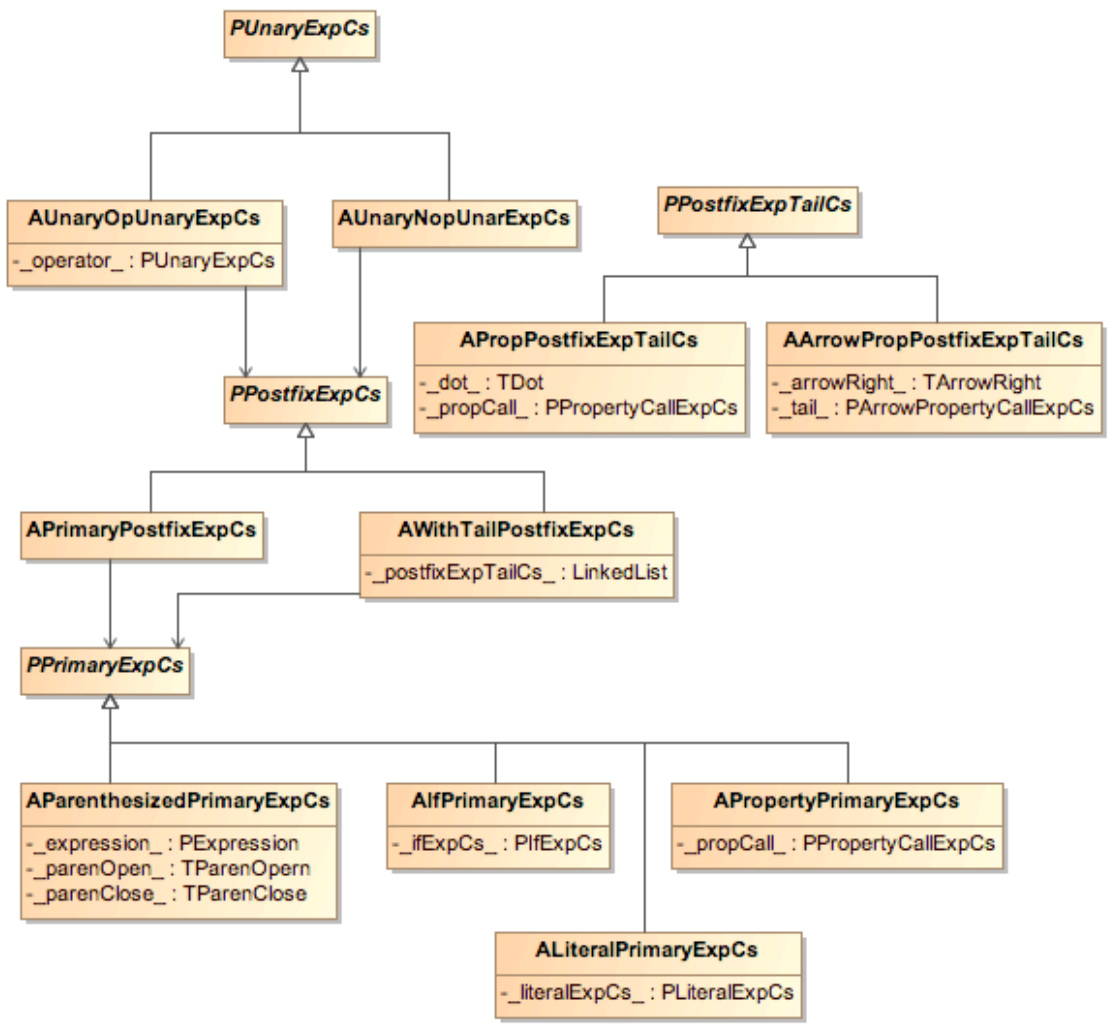

PUnaryExpCs
AUnaryOpUnaryExpCs
-_operator_ : PUnaryExpCs
AUnaryNopUnarExpCs
PPostfixExpTailCs
APropPostfixExpTailCs
-_dot_ : TDot
-_propCall_ : PPropertyCallExpCs
AArrowPropPostfixExpTailCs
-_arrowRight_ : TArrowRight
-_tail_ : PArrowPropertyCallExpCs
PPostfixExpCs
APrimaryPostfixExpCs
AWithTailPostfixExpCs
-_postfixExpTailCs_ : LinkedList
PPrimaryExpCs
AParenthesizedPrimaryExpCs
-_expression_ : PExpression
-_parenOpen_ : TParenOpern
-_parenClose_ : TParenClose
AIfPrimaryExpCs
-_ifExpCs_ : PIfExpCs
APropertyPrimaryExpCs
-_propCall_ : PPropertyCallExpCs
ALiteralPrimaryExpCs
-_literalExpCs_ : PLiteralExpCs

Figure 13.12: SableCC-Ext generated classes for unary expressions

The diagram before represents the generated classes for the grammar rules explained at Definition 13.12

13.3.4 Property Call Expressions

Following our explanation next section is dedicated to property call expressions. First alternative of *property_call_exp_cs* production is a path name with a time expression. It is used to access attributes or association ends which name matches with path name. Time expressions are the elements that contain the @pre directive.

```
property_call_exp_cs <OclExpression> =
     {path_time} <OclExpression> [name]:path_name_cs
       [time]:time_exp_cs?    #nocreate
    |{arg_list}  <OclExpression> [name]:path_name_cs
       [time]:time_exp_cs? parameters]:property_call_parameters_cs
       #customheritage #nocreate
    |{qualified} <OclExpression> [name]:path_name_cs
      [qualifiers]:qualifiers #customheritage [time]:time_exp_cs?
      #nocreate  ;

property_call_parameters_cs <List> =
      paren_open [param_list]:actual_parameter_list_cs?
      #customheritage paren_close #nocreate  ;
```

Definition 13.13: Grammar rules for property call expressions

Second alternative is a path name with parameters. It can be used to call operations with parameters. And finally, last alternative use qualifiers to access elements.

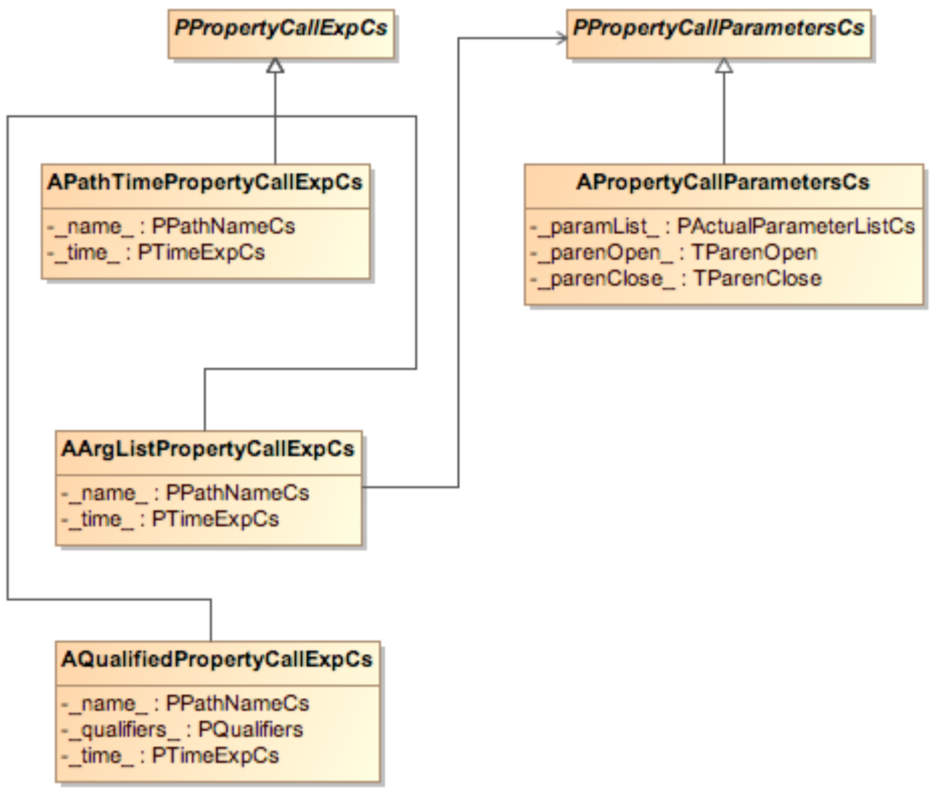

PPropertyCallExpCs
PPropertyCallParametersCs
APathTimePropertyCallExpCs
-_name_ : PPathNameCs
-_time_ : PTimeExpCs
APropertyCallParametersCs
-_paramList_ : PActualParameterListCs
-_parenOpen_ : TParenOpen
-_parenClose_ : TParenClose
AArgListPropertyCallExpCs
-_name_ : PPathNameCs
-_time_ : PTimeExpCs
AQualifiedPropertyCallExpCs
-_name_ : PPathNameCs
-_qualifiers_ : PQualifiers
-_time_ : PTimeExpCs

Figure 13.13: SableCC-Ext generated classes for property call expressions

13.3.4.1 Qualifiers

Qualifiers productions are common lists of *simple_name* tokens separated by commas. We can use more than one qualifier in a qualified property call expression although in our processing of a qualified property call expression we only support one qualifier for binary relationships with association class and the same class owner for the association ends.

```
qualifiers <List> =
      bracket_open qualifiers_list_cs #customheritage
      bracket_close #chain ;

qualifiers_list_cs <List> =
      [element]:qualifiers_list_element_cs
      [tail]:qualifiers_list_tail_cs? #nocreate ;

qualifiers_list_element_cs <String> =
    {qualifier}simple_name #chain ;

qualifiers_list_tail_cs <List> =
        comma [tail]:qualifiers_list_cs #chain ;
```

Definition 13.14: Grammar rules for qualifiers

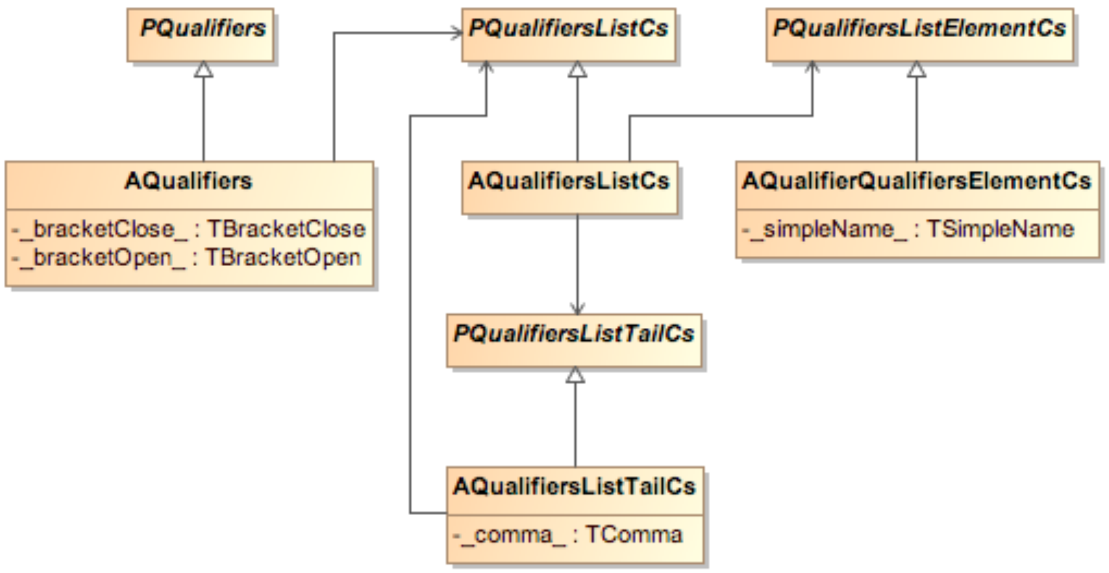


Figure 13.14: SableCC-Ext generated classes for qualifiers

13.3.5 Arrow Expressions

Arrow expressions are such expressions that are preceded by an arrow -> token. There are three alternatives inside this production.

```
arrow_property_call_exp_cs <OclExpression> =
     {iterate} <IterateExp>
         T.iterate paren_open
         [iterators]:iterate_vars_cs? #customheritage
         [accumulator]:initialized_variable_cs
         vertical_bar [body]:expression #customheritage
         paren_close
    | {iterator} <IteratorExp>
         [name]:iterator_name_cs paren_open
         [iterators]:iterator_vars_cs? #customheritage
         [body]:expression #customheritage paren_close
    | {operation} <OperationCallExp>
         [name]:simple_name paren_open
          [parameters]:actual_parameter_list_cs? #customheritage
          paren_close
    ;
```

Definition 13.15: Grammar rules for arrow expressions

First alternative is an iterate constructions. As we explained in earlier chapters, iterate constructions begins with the *iterate* keyword followed by an expression between parentheses including iterate variables, an accumulator variable, and a body expression preceded by a vertical bar. As indicated, it results in an IterateExp object.

Second alternative is an iterator construction that begins with a iterator name (e.g., forAll, one, select, reject, …) followed by an expression between parentheses including iterator variables, with a vertical bar inside, and a body expression. As indicated, it results in an IteratorExp object.

Last alternative is a collection operation including a list of actual parameters. As indicated, it results in an OperationCallExp object.

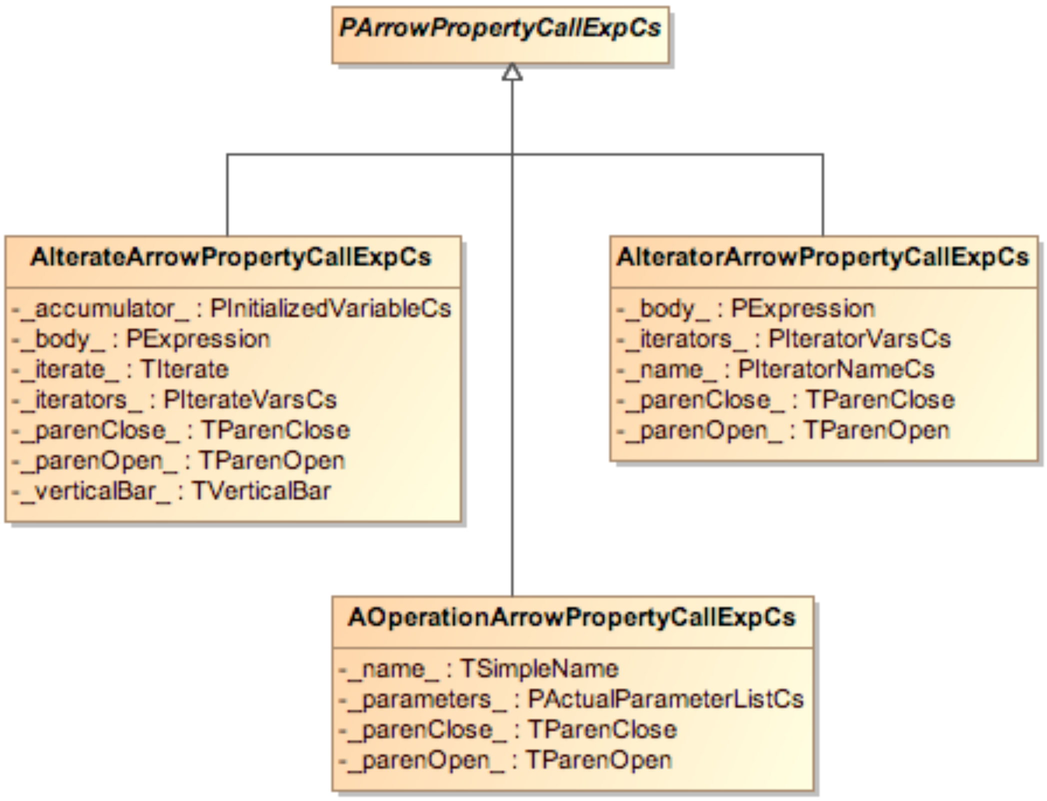

PArrowPropertyCallExpCs
AIterateArrowPropertyCallExpCs
-_accumulator_ : PInitializedVariableCs
-_body_ : PExpression
-_iterate_ : TIterate
-_iterators_ : PIterateVarsCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen
-_verticalBar_ : TVerticalBar
AIteratorArrowPropertyCallExpCs
-_body_ : PExpression
-_iterators_ : PIteratorVarsCs
-_name_ : PIteratorNameCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen
AOperationArrowPropertyCallExpCs
-_name_ : TSimpleName
-_parameters_ : PActualParameterListCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen

Figure 13.15: SableCC-Ext generated classes for arrow expressions

13.3.6 TIME EXPRESSIONS

Time expressions productions provide the usage of the @pre directive to be used in postconditions.

```
time_exp_cs <OclTimeExp> =
        is_marked_pre_cs #chain
    ;

is_marked_pre_cs <OclTimeExp> =
        at_pre #nocreate
    ;
```

Definition 13.16: Grammar rules for time expressions

The result of these productions is an OclTimeExp object, which is part of the additional classes of Dresden OCL2 Toolkit that simplifies the parsing process.

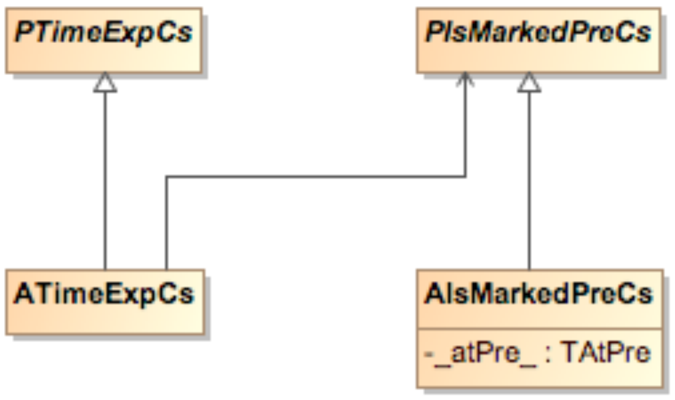


Figure 13.16: SableCC-Ext generated classes for time expressions

13.3.7 Variables

In this section we will present the set of productions that recognize variables for to use in the OCL 2.0 language.

13.3.7.1 Iterate variables

The *iterate_vars_cs* production specifies the first part of an iterate constructions that consists in a list of variables followed by a semi colon sign. Such variables share the same production than actual parameters due to they are syntactically identical.

```
iterate_vars_cs <List> =
      [iterators]:actual_parameter_list_cs #customheritage
      semi_colon #chain
   ;
```

Definition 13.17: Grammar rule for iterate variables

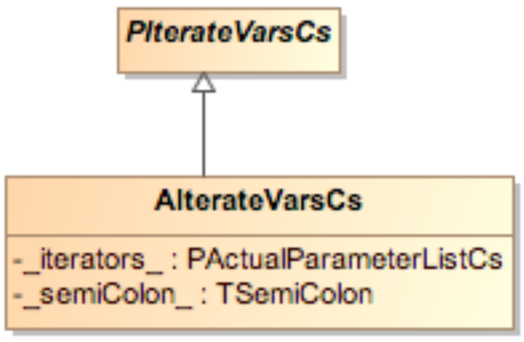

PIterateVarsCs
AIterateVarsCs
-_iterators_ : PActualParameterListCs
-_semiColon_ : TSemiColon

Figure 13.17: SableCC-Ext generated classes for iterate variables

13.3.7.2 Iterator variables

The *iterator_vars_cs* production specifies the first part of a iterator expression that consists in a list of variables followed by a vertical bar that separates these variables from the iterator body expression. These variables are specified by the same production than actual parameters because they are syntactically identical.

```
iterator_vars_cs <List> =
      [iterators]:actual_parameter_list_cs #customheritage
      vertical_bar
      #chain
    ;
```

Definition 13.18: Grammar rule for iterator variables

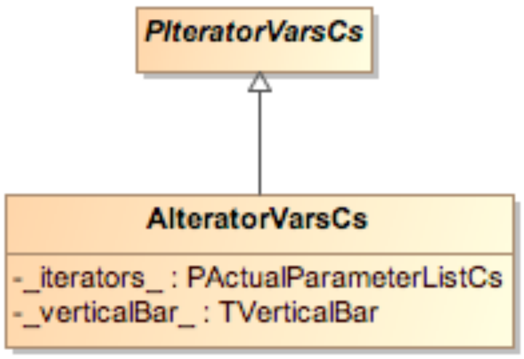

PIteratorVarsCs
AIteratorVarsCs
-_iterators_ : PActualParameterListCs
-_verticalBar_ : TVerticalBar

Figure 13.18: SableCC-Ext generated classes for iterator variables

13.3.7.3 Initialized variables

Initialized variables are used to represent variables in different situations like for example inside a let expression. These productions specify a list of variables separated by commas. Each element of such list consists in a name and type (like a formal parameter), and an expression that initializes the element (or variable).

```
initialized_variable_list_cs <List> =
        [item]:initialized_variable_cs
        [tail]:initialized_variable_list_tail_cs* #nocreate  ;

initialized_variable_list_tail_cs <Variable> =
        comma [item]:initialized_variable_cs #chain  ;

initialized_variable_cs <Variable> =
        [name_and_type]:formal_parameter_cs
        [initializer]:variable_initializer  ;

variable_initializer <OclExpression> =
        !equals [init_value]:ocl_expression_cs #chain  ;
```

Definition 13.19: Grammar rules for initialized variables

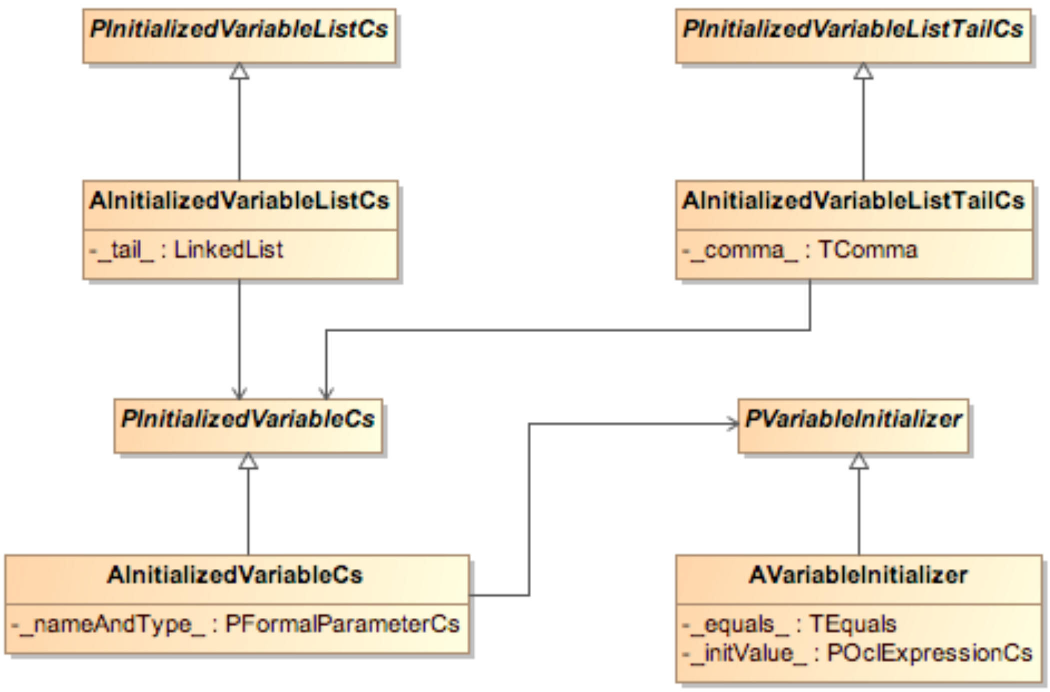

PInitializedVariableListCs
PInitializedVariableListTailCs
AInitializedVariableListCs
-_tail_ : LinkedList
AInitializedVariableListTailCs
-_comma_ : TComma
PInitializedVariableCs
PVariableInitializer
AInitializedVariableCs
-_nameAndType_ : PFormalParameterCs
AVariableInitializer
-_equals_ : TEquals
-_initValue_ : POclExpressionCs

Figure 13.19: SableCC-Ext generated classes for initialized variables

13.3.8 Parameters

In this section we will introduce the productions that recognizes list of parameters that can be used in the specification of an operation header or directly in an operation call.

13.3.8.1 Actual parameters

Actual parameters are the parameters used in an operation call. These productions represent a list of parameters separated by commas. Each parameter can be directly an expression or a formal parameter depending on if the type specification is needed or not.

```
actual_parameter_list_cs <List> =
       [element]:actual_parameter_list_element_cs
       [tail]:actual_parameter_list_tail_cs? #nocreate
    ;

actual_parameter_list_tail_cs <List> =
        comma [tail]:actual_parameter_list_cs #chain
    ;

actual_parameter_list_element_cs <OclActualParameterListItem> =
      {untyped} [element]:expression
    | {typed}   [param]:formal_parameter_cs
    ;
```

Definition 13.20: Grammar rules for actual parameters

When we use an operation inside an OCL expression we use the untyped branch because the parameters are not formal. They are real parameters.

It is important to note here the usage of the additional class *OclActualParameterListItem* to contain the resultant object of the *actual_parameter_list_element_cs* production.

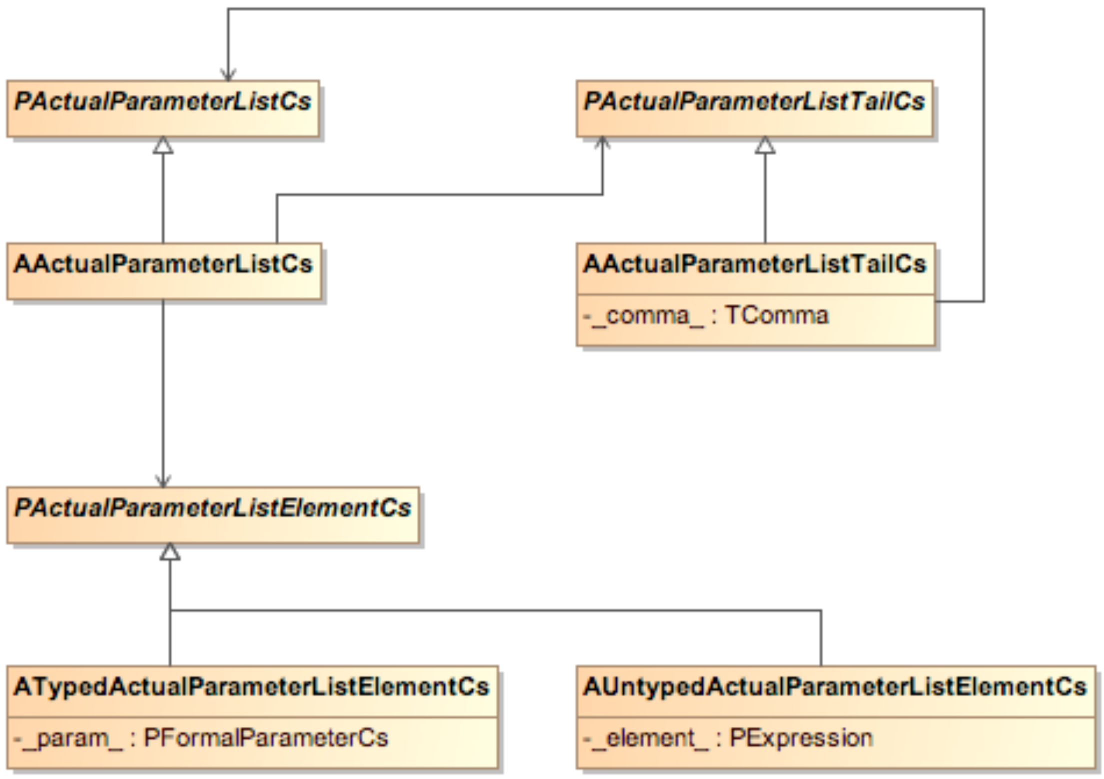

PActualParameterListCs
PActualParameterListTailCs
AActualParameterListCs
AActualParameterListTailCs
-_comma_ : TComma
PActualParameterListElementCs
ATypedActualParameterListElementCs
-_param_ : PFormalParameterCs
AUntypedActualParameterListElementCs
-_element_ : PExpression

Figure 13.20: SableCC-Ext generated classes for actual parameters

13.3.8.2 Formal parameters

Formal parameters are the other kind of parameters that have to been recognised by our OCL 2.0 grammar.

These productions provide the specification of a list of parameters separated by commas. Each parameter consists in a simple name followed by a colon and a type specifier.

It is important to note that here we use Classifier metaclass to specify the return object of a *formal_parameter_type_specifier* production instead of Type metaclass. We maintain Classifier because Dresden grammar use Classifier and due to although Classifier is an instance of Type, all types that can be used in our grammar are indirectly also subclasses of Classifier, so to use Classifier is not a limitation.

```
formal_parameter_cs <OclFormalParameter> =
        [name]:simple_name
        [type]:formal_parameter_type_specifier
    ;

formal_parameter_type_specifier <Classifier> =
        colon [type]:type_specifier #chain
    ;

formal_parameter_list_cs <List> =
        [item]:formal_parameter_cs
        [tail]:formal_parameter_list_tail_cs? #nocreate
    ;

formal_parameter_list_tail_cs <List> =
        comma [tail]:formal_parameter_list_cs #chain
    ;
```

Definition 13.21: Grammar rules for formal parameters

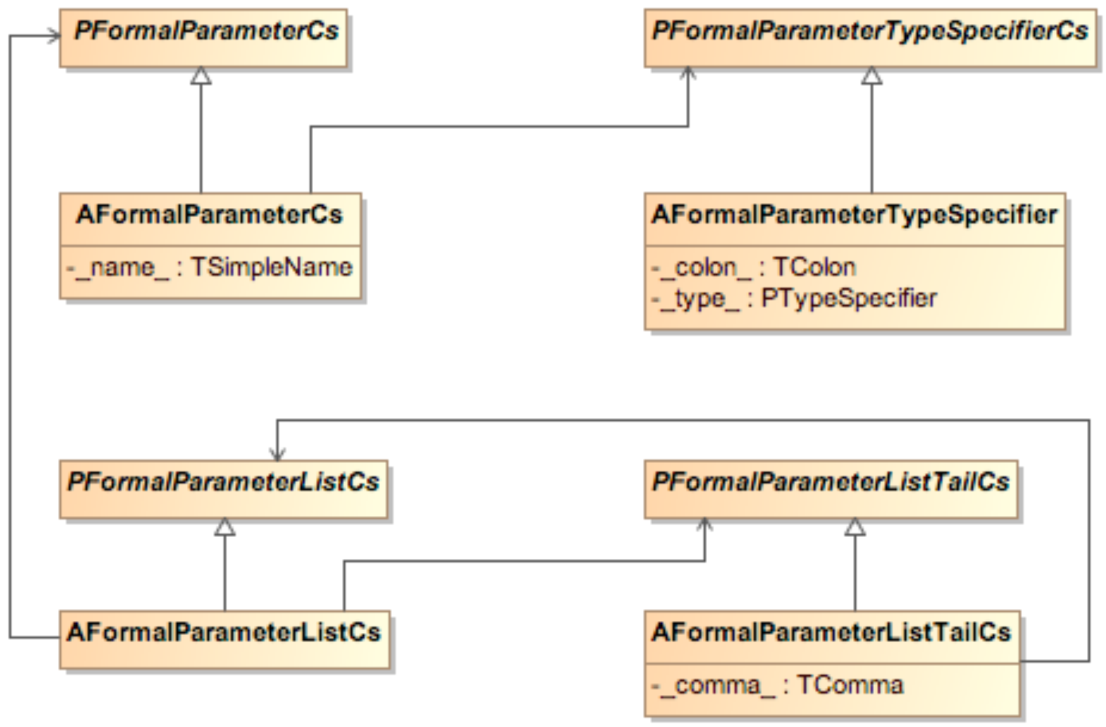


Figure 13.21: SableCC-Ext generated classes for formal parameters

13.3.9 Types

This section introduces the productions of our grammar that provide specification of types in OCL expressions.

```
type_specifier <Classifier> =
     {simple_type} simple_type_specifier_cs #chain #nocreate
   | {collection_type} collection_type_specifier_cs #chain
       #nocreate
   | {tuple_type} tuple_type_specifier_cs #chain #nocreate
    ;
```

Definition 13.22: Grammar rules for types

We have three alternatives inside the production rule. First one recognizes simple types, like Integer, Boolean or String. Last two alternatives recognise collection types and tuple types respectively.

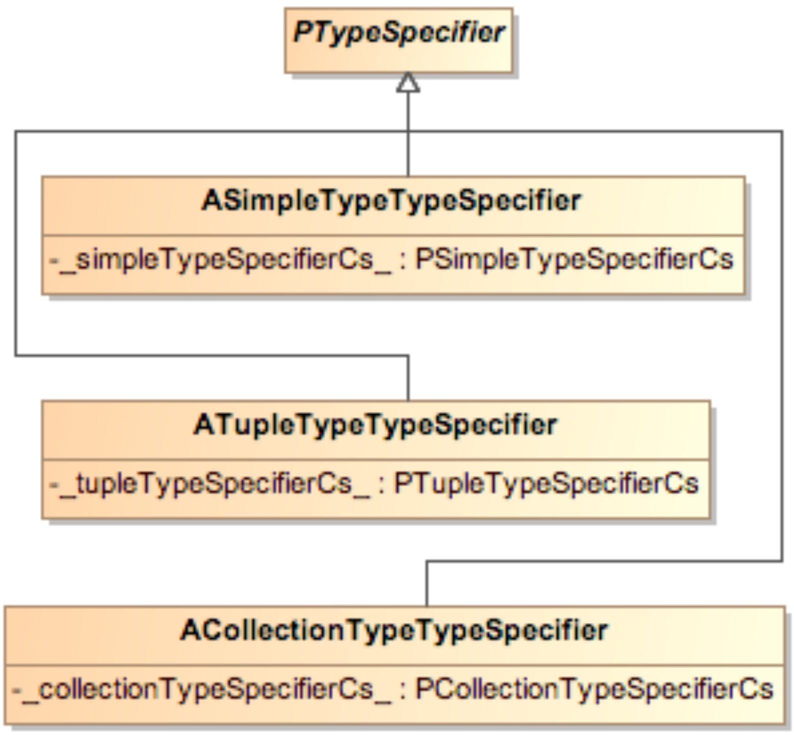

PTypeSpecifier
ASimpleTypeTypeSpecifier
-_simpleTypeSpecifierCs_ : PSimpleTypeSpecifierCs
ATupleTypeTypeSpecifier
-_tupleTypeSpecifierCs_ : PTupleTypeSpecifierCs
ACollectionTypeTypeSpecifier
-_collectionTypeSpecifierCs_ : PCollectionTypeSpecifierCs

Figure 13.22: SableCC-Ext generated classes for types

13.3.9.1 Simple types

Simple types production is very general because includes only a path name as alternative. This simplicity implies that the semantic checking will be more difficult although it is better than a more complex grammar.

```
simple_type_specifier_cs <Classifier> =
        path_name_cs #nocreate
    ;
```

Definition 13.23: Grammar rule for simple types

With this production we can recognise primitive types like Integer, String, Real or Boolean, and also UML classes or enumeration types, which already are valid types.

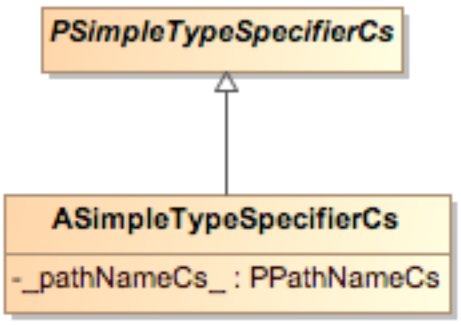


Figure 13.23: SableCC-Ext generated classes for simple types

13.3.9.2 Collection types

Collection type's production is very simple. It consists in a keyword specifying the collection type (e.g., Set, Bag, OrderedSet and Sequence) followed by another type specified between parentheses.

Here we do not allow CollectionType as a valid keyword to specify a collection type kind because of CollectionType class in the OCL metamodel is abstract, therefore it must not have instances.

```
collection_type_specifier_cs <CollectionType> =
        [kind]:collection_type_identifier_cs
        paren_open [type]:type_specifier paren_close #nocreate
    ;
```

Definition 13.24: Grammar rule for collection types

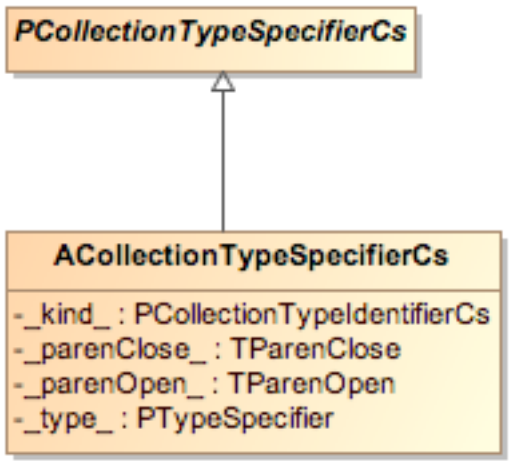

PCollectionTypeSpecifierCs
ACollectionTypeSpecifierCs
-_kind_ : PCollectionTypeIdentifierCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen
-_type_ : PTypeSpecifier

Figure 13.24: SableCC-Ext generated classes for collection types

13.3.9.3 Tuple types

Tuple types' productions are also very simple because only consists in the TupleType token followed by a list of formal parameters between parentheses.

Each element inside a tuple type declaration uses the same production as formal parameter because they are syntactically identical. They are formed with a name and a type.

```
tuple_type_specifier_cs <TupleType> =
        tuple_type paren_open
        [tuple_members]:formal_parameter_list_cs?
        paren_close #nocreate
    ;
```

Definition 13.25: Grammar rule for tuple types

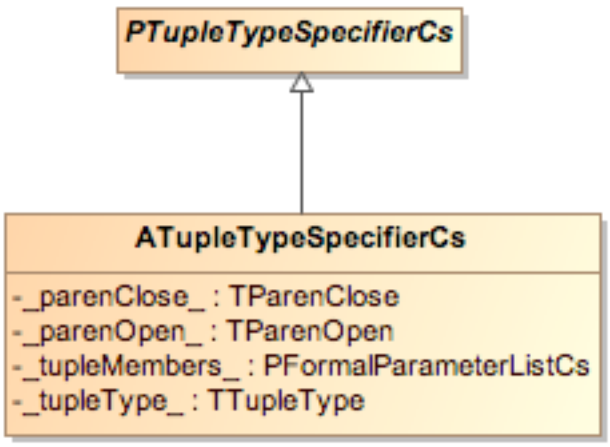

PTupleTypeSpecifierCs
ATupleTypeSpecifierCs
-_parenClose_ : TParenClose
-_parenOpen_ : TParenOpen
-_tupleMembers_ : PFormalParameterListCs
-_tupleType_ : TTupleType

Figure 13.25: SableCC-Ext generated classes for tuple types

13.3.10 LITERAL EXPRESSIONS

The final section of our OCL 2.0 grammar is dedicated to literal expressions. As we explained in earlier chapters, literal expressions are divided in three groups. First one is the group of the primitive literals including Integer, Real, String and Boolean constant values.

The other groups are the collection and tuple literals, which are very useful in the OCL 2.0 language.

```
literal_exp_cs <LiteralExp> =
     {lit_primitive}    primitive_literal_exp_cs    #chain
   | {lit_collection}   collection_literal_exp_cs   #chain
   | {lit_tuple}        tuple_literal_exp_cs        #chain
   ;
```

Definition 13.26: Grammar rule for literal expressions

As you can see, the production rule here is very simple because it only contains an alternative for one of these groups.

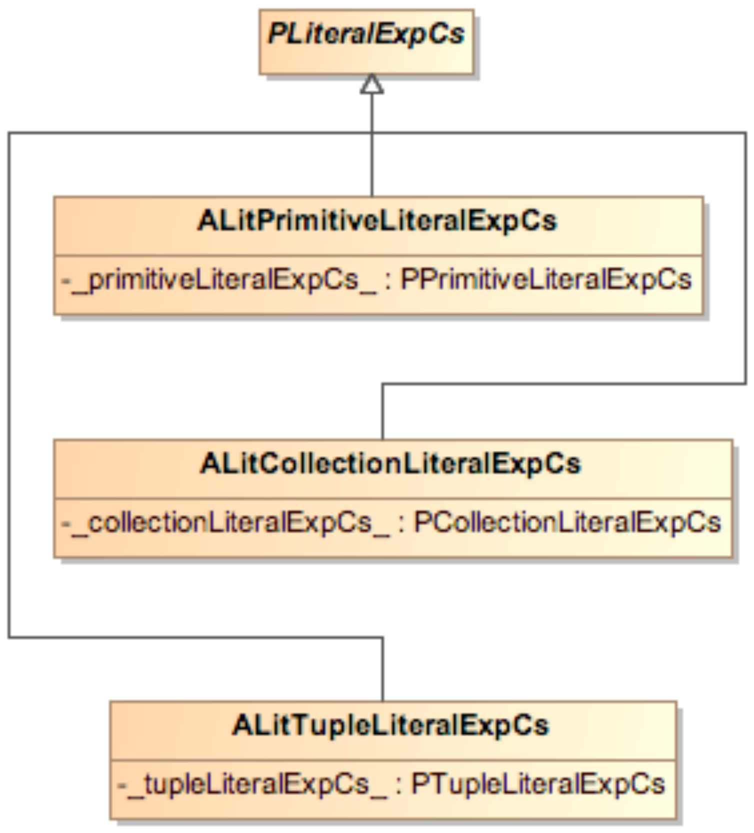

PLiteralExpCs
ALitPrimitiveLiteralExpCs
-_primitiveLiteralExpCs_ : PPrimitiveLiteralExpCs
ALitCollectionLiteralExpCs
-_collectionLiteralExpCs_ : PCollectionLiteralExpCs
ALitTupleLiteralExpCs
-_tupleLiteralExpCs_ : PTupleLiteralExpCs

Figure 13.26: SableCC-Ext generated classes for literal expressions

13.3.10.1 Primitive literals

Primitive literals are divided into three groups. First one includes numeric literal expressions as written in the *primitive_literal_exp_cs* production of our grammar.

Such first alternative is also divided in two branches: integer and real literals. Each one contains a token element that recognises Integer or Real values. The return object of such productions is an IntegerLiteralExp or a RealLiteralExp. Both are metaclasses of the OCL 2.0 metamodel.

Second alternative inside primitive literals' production is the string alternative that recognises String values and returns an instance of StringLiteralExp metaclass.

And finally, the last alternative is for to recognise boolean values like true or false.

```
primitive_literal_exp_cs <PrimitiveLiteralExp> =
      {numeric} numeric_literal_exp_cs  #chain
    | {string} string_literal_exp_cs    #chain
    | {boolean} boolean_literal_exp_cs  #chain
    ;

numeric_literal_exp_cs <NumericLiteralExp> =
      {int}  <IntegerLiteralExp> [integer]:integer_literal
    | {real} <RealLiteralExp>    [real]:real_literal
    ;

string_literal_exp_cs <StringLiteralExp> =
        [value]:string_literal
    ;

boolean_literal_exp_cs <BooleanLiteralExp> =
      {false} false #nocreate | {true} true #nocreate
    ;
```

Definition 13.27: Grammar rules for primitive literals

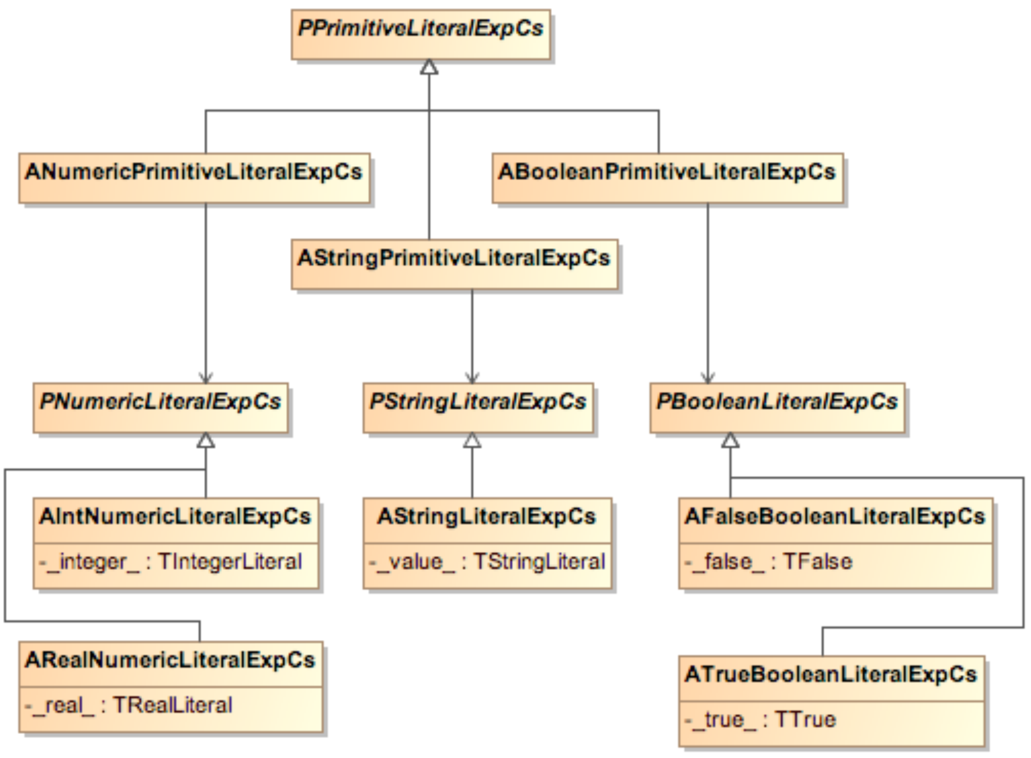


Figure 13.27: SableCC-Ext generated classes for primitive literals

13.3.10.2 Tuple literals

To recognise tuple literals by means of production rules is very simple here because we only need a Tuple token followed by a list of tuple elements between curly brackets.

Note that we do not reuse *initialized_variable_list_cs* production because although elements inside a tuple literal expression are syntactically identical to initialized variables, the type specifier of these elements is optional. Each tuple element has a name, a type and an expression that defines its value.

```
tuple_literal_exp_cs <TupleLiteralExp> =
        tuple brace_open
        [tuple_part]:tuple_part_list_cs
        brace_close
    ;
tuple_part_list_cs <List> =
        [item]:tuple_part_cs
        [tail]:tuple_part_list_tail_cs* #nocreate
    ;

tuple_part_list_tail_cs <Variable> =
        comma
        [item]:tuple_part_cs #chain
    ;

tuple_part_cs <Variable> =
        [name]:simple_name
        tuple_part_type?
        [initializer]:variable_initializer
    ;

tuple_part_type <Classifier> =
        !colon [type]:type_specifier #chain
    ;
```

Definition 13.28: Grammar rules for tuple literals

It is important to note that inside the tuple_part_cs production we can see the type of the member of a tuple literal expression is syntactically optional.

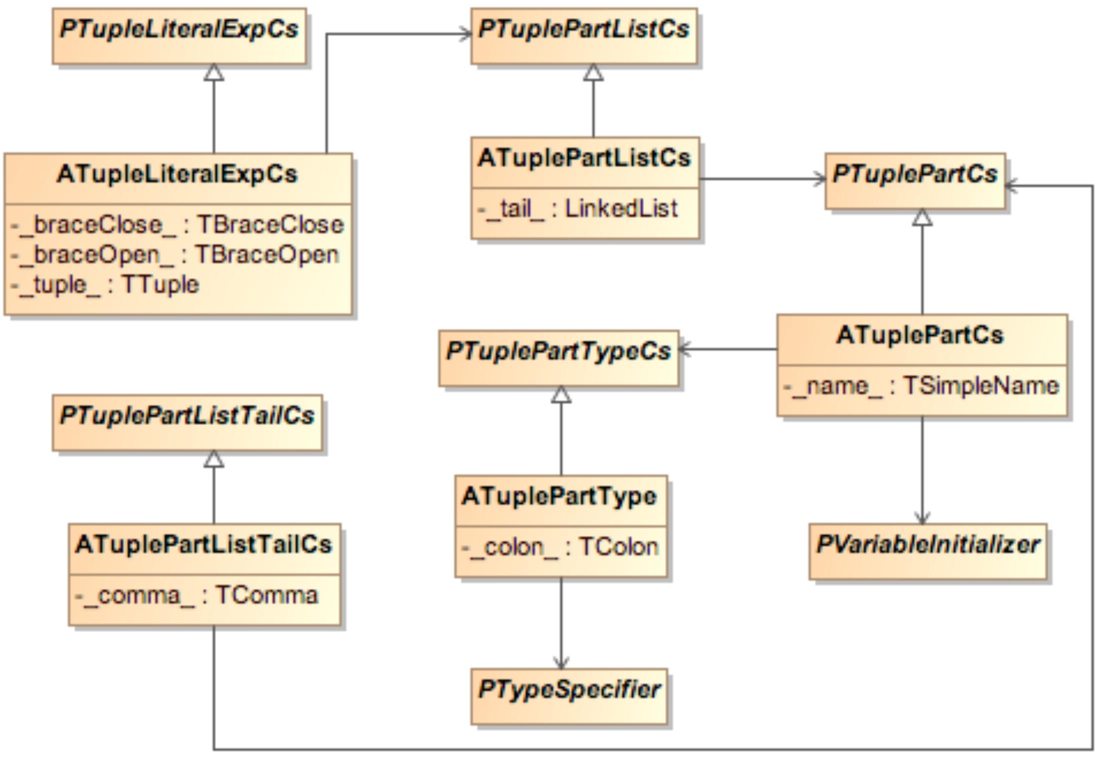


Figure 13.28: SableCC-Ext generated classes for tuple literals

In the previous model we can see the generated classes for the tuple literals productions. As we can compare with the generated classes for an initialized variable list, they are very similar.

The relation between ATuplePartCs and PTuplePartTypeCs has a cardinality of 0 or 1 because, as was explained earlier, when writing a tuple part it is optional to specify it type.

13.3.10.3 Collection literals

Last elements to recognise are collection literals. It is important to remember here that we can create a collection literal through a list of elements with the same type or through a range of values.

First of all we can see *collection_literal_exp_cs* production that recognises a collection kind identifier (Set, Bag, OrderedSet or Sequence) followed by a list of collection elements between curly brackets.

```
collection_literal_exp_cs <CollectionLiteralExp> =
        [kind]:collection_type_identifier_cs
        brace_open
        [parts]:collection_literal_parts_cs?
        brace_close
    ;

collection_literal_parts_cs <List> =
        [part]:collection_literal_part_cs
        [tail]:collection_literal_parts_tail_cs? #nocreate
    ;

collection_literal_parts_tail_cs <List> =
        comma [tail]:collection_literal_parts_cs #chain
    ;

collection_literal_part_cs <CollectionLiteralPart> =
        {range}      collection_range_cs #chain
    |   {single_exp} <CollectionItem> expression
    ;

collection_range_cs <CollectionRange> =
        [first]:expression dbl_dot [last]:expression
    ;
```

Definition 13.29: Grammar rules for collection literals

Next production rules implement this list of elements separated by commas and finally *collection_literal_part_cs* production specifies how an element is made. Each element can be an expression or a range of values that consists in two expressions separated by a double dot.

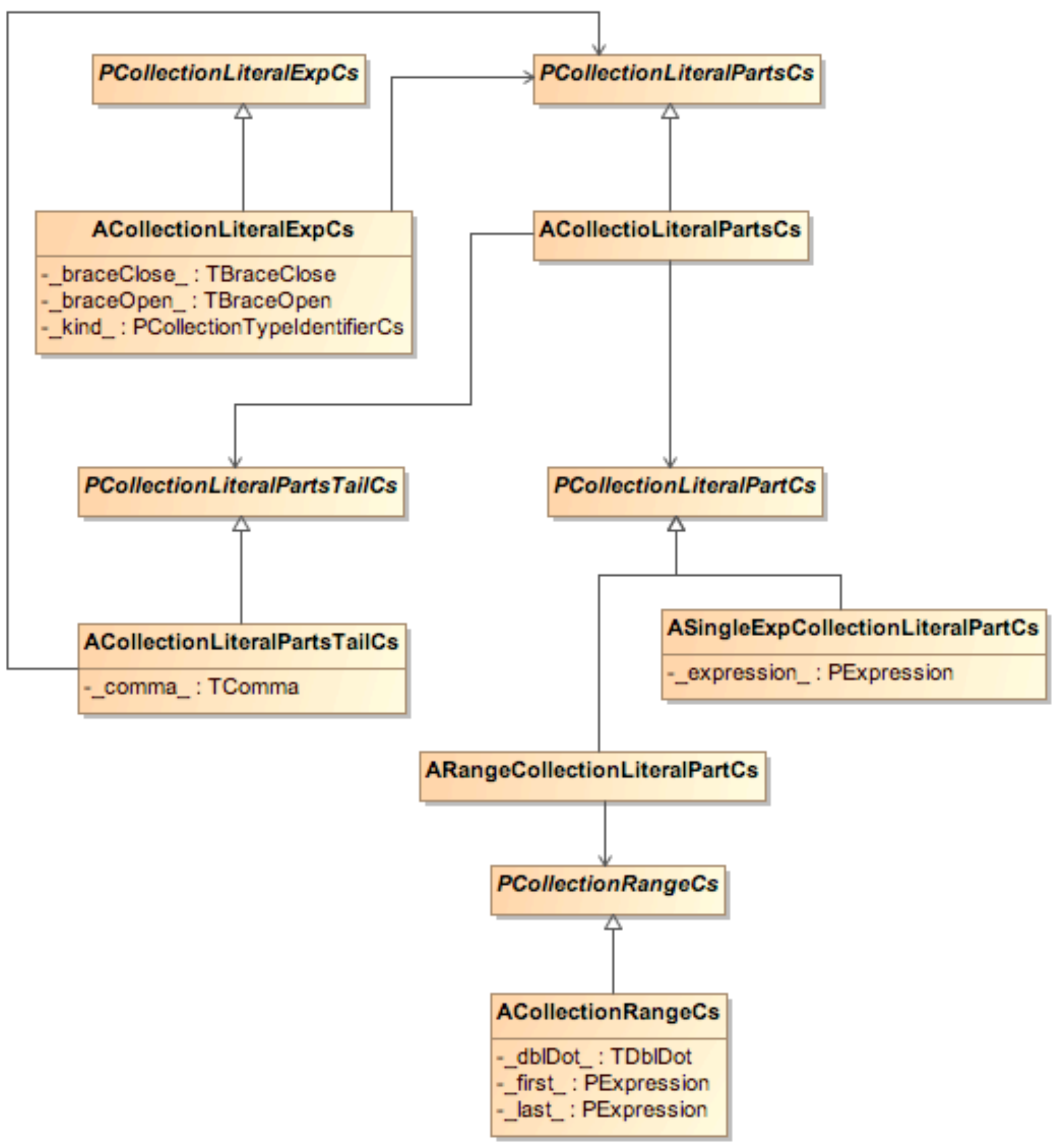

PCollectionLiteralExpCs
PCollectionLiteralPartsCs
ACollectionLiteralExpCs
-_braceClose_ : TBraceClose
-_braceOpen_ : TBraceOpen
-_kind_ : PCollectionTypeIdentifierCs
ACollectioLiteralPartsCs
PCollectionLiteralPartsTailCs
PCollectionLiteralPartCs
ASingleExpCollectionLiteralPartCs
-_expression_ : PExpression
ACollectionLiteralPartsTailCs
-_comma_ : TComma
ARangeCollectionLiteralPartCs
PCollectionRangeCs
ACollectionRangeCs
-_dblDot_ : TDblDot
-_first_ : PExpression
-_last_ : PExpression

Figure 13.29: SableCC-Ext generated classes for collection literals

DEVELOPMENT PROCESS 14

14 DEVELOPMENT PROCESS

14.1 AN ITERATIVE APPROACH

The development of our OCL expressions processor follows an iterative process. As we briefly introduced in the first chapter these are the steps to make our parser:

1. Select a subset of the OCL 2.0 language and study it.

2. Write a specification file for SableCC-Ext including both tokens and productions in order to recognize the language of the previous subset.

3. Invoke SableCC-Ext with such specification file in order to obtain the generated framework that will be useful to do the parsing process.

4. Complete the necessary methods of the skeleton walker class (LAttrEvalAdapter.java file) with the correct instructions to provide the instantiation of OCL expressions as metamodel instances.

5. Test the behaviour of the parser with some examples in order to verify a correct functionality.

6. Go back to first step until our parser supports all OCL 2.0 language.

At this chapter we will explain some changes made to the Parser Subsystem of the Dresden OCL2 Toolkit to make easier the development of our processor. Then, we will show an example of development iteration with the addition of a new hypothetical OCL statement to our parser.

14.2 FIRST STEPS

Since we have based the design of our processor on the Dresden OCL2 Toolkit, we began with a release package of such tool and we adapt it until to obtain a new structure of files that become in our processor. At this section we will introduce some of the changes made in order to structure the code and adapt it to our iterative development process.

14.2.1 ADAPTING SABLECC-EXT

SableCC-Ext comes inside the release package of the Dresden OCL2 Toolkit. We decided to extract it and to have it as an independent tool in its own release package. To do this task we had to study which were the union points between this tool and Dresden tool in order to separate them in a correct way.

After to convert SableCC-Ext into a standalone tool we have the next package hierarchy:

- **grammars**: inside this directory we find the specification files for SableCC-Ext. Concretely we are interested in the gmcOCL202.xgrammar file, which contains all our OCL 2.0 grammar. Such grammar is based on the Dresden OCL grammar although it has several changes because Dresden one contains some rules that are related to the UML 1.5 version instead of the 2.0.

- **lib**: contains all libraries needed to use SableCC and SableCC-Ext including antlr tool an more.

- **obj**: contains the structure of Java binary files (.class) needed to work with SableCC-Ext. It is important here the subdirectory obj.jmi-api.uml2 that owns the binary classes of the metamodel classes used in EinaGMC. This is a change in SableCC-Ext standalone version due to adapt it to our conceptual modeling environment used.

- **src**: contains the source Java classes (.java). The generated framework of our grammar is placed here inside the directory src.parser.sablecc.

- **build.xm**l: is the file containing the ant instruction to generate our framework through SableCC-Ext and our OCL 2.0 grammar.

It is important to note that to generate the framework classes of our processor we have to use the next ant command in a terminal with the ant tool installed being located in the base directory of the SableCC-Ext release directory:

```
~/> ant compiler.generation
```

Another important concept is that we had to change two special files inside SableCC-Ext package structure. These files are *src.org.sablecc.sablecc.analyses.txt* and *src.org.sablecc.sablecc.TypeMap.java* both inside the *src.org.sablecc.sablecc* subdirectory.

First one contains templates for the code generation of the framework classes. Such templates have a problem with the imports. The import statements are hard-wired inside the code generation templates and they references classes of the Dresden OCL2 Toolkit. Therefore we had to study such file and change these imports properly with the Eina GMC version of the imported classes.

And finally, second file maps type names to package names of that type. Used in generator for LAttrEvalAdapter (our generated skeleton walker class) to allow reflective access to type system and type checks, it contained references to classes of Dresden OCL2 Toolkit that were changed for Eina GMC environment classes.

14.2.2 Adapting the Parser Subsystem

After separating SableCC-Ext from the whole release package of Dresden OCL2 Toolkit we had to adapt some parts of the framework of the Parser Subsystem in order to make it compatible with the new generated classes from SableCC-Ext connected with the Eina GMC conceptual modeling environment.

Here we present the most significant changes made to adapt the Parser Subsystem to the Eina GMC. Note that we will not introduce the final structure of our processor here due to it will be explained at next chapter. We only want to show important elements that were changed.

14.2.2.1 Node factory

The Node Factory (NodeFactory.java) is one of the most important elements in the Dresden OCL2 Toolkit because it provides the *createNode(String type)* method. Such method returns an object whose type is specified by the input parameter String. For example, if we want to obtain a List, we only have to call such method with the String "List" as a parameter.

This factory is based on a hash table of pairs <Type name, constructor of such type>. So, when a name is found, the factory calls the method to create such class type.

The Node Factory was initialized with pairs on its hash table specifying classes of the Dresden metamodel. Therefore, we changed this initialization with new pairs identifying the classes of the UML and OCL metamodels of the Eina GMC environment. As we explained when we introduced Eina GMC, every class in the Eina GMC metamodels have a facade class that provides a public creator method for each element within it. So, we changed the Dresden classes and methods for these ones to make a correct adaptation to our tool.

This way we have now a Node Factory that is related to the Eina GMC and is used in the same way than before.

14.2.2.2 AST Generator

The AST Generator is the class that implements such methods that are not completed in the skeleton class provided by SableCC-Ext. Dresden provides a class with all these methods implemented, although it is very related to its own metamodel classes. Furthermore, such classes implements the UML metamodel in its 1.5 version and we need a 2.0 adaptation.

Due to these problems, we decided to begin our own implementation of such methods in a new AST Generator class different than Dresden one. We thought that directly change the Dresden AST Generator will be a tedious and error-prone task so to begin a new implementation from the ground up here was the best decision.

Our AST Generator class is inside the LAttrGMCAstGenerator.java file. First of all we decided to begin with a minimal subset of the OCL 2.0 language and we write a basic specification file with only invariant constraints and relational expression with Integers. So we only were able to parse constraints like the following:

```
context Person inv: 1<9
```

Example 14.1: Very simple OCL constraint

The attribute evaluation skeleton class generated for the grammar that we wrote needed between a ten and a twenty of methods to provide the semantic checking and the metamodel instantiation of such expressions.

Due to the result of these processes was satisfactory, we opted to follow this iterative method and to add more complexity to the grammar and to the AST Generator file until complete the whole OCL 2.0 language.

14.2.2.3 Type checker

The Type checker of the Parser Subsystem of the Dresden OCL2 Toolkit is an element that works in a second sweep of the parse tree.

We decided to make our own type checker element in the same sweep than the parsing process. To do this we create a new Java source class called TypeCheck.java that implements a reflective visitor. This visitor uses the Java Reflection API that provides the functionality of discover the java class of a java object and to access to its attributes and methods. With all of it we are able to determine the correct class of a Java Object, therefore we can implement a type checker method for every element in the OCL metamodel. Furthermore, we only have to call the same method with the element to check as a parameter. Our reflective visitor is able to determine the class of such parameter and then invoke the correct checker method.

With this design idea we simplify the implementation of the Type Checker with a divide and conquer strategy that consists in to implement only the type checker method of each element in the OCL metamodel separately.

Finally we named the type checker global method as *typecheck* whose parameter is an element of the metamodel. Calling such method we indirectly call the particular method written to check the parameter according to its class.

14.3 A NEW ITERATION: SWITCH-CASE

Imagine that the Object Management Group (OMG), which manages both the UML and OCL modeling languages, decides to add a new expression statement to the set of expressions of the Object Constraint Language 2.0. Also imagine that such new statement is a common switch-case construction that is present in most programming languages.

With this imaginary context we will explain how to make a complete development iteration of our processor of OCL 2.0 expressions.

14.3.1 A NEW PROPOSAL

First of all, to understand the new language statement and its metamodel instantiation is mandatory.

As we can see below a switch-case statement contains three different parts. First one consists on a switch keyword followed by an expression between parentheses. Such expression is the element that we will evaluate.

Next section of a switch-case statement is formed by at least one case section. Each case section consists on a case keyword and two expressions separated by a then keyword. First expression here is a possible value of the previous element expression to evaluate, and second one consists on an expression that will be executed only if the value expression matches with such element expression executed. It is important to note that each case section ends with an endcase keyword.

Finally, last section is the default expression that will be executed only if none of the value expressions of the case sections matches with the value of the first element expression.

```
switch (element-oclexpression)
  case value-oclexpression then result-oclexpression endcase
  case value-oclexpression then result-oclexpression endcase
   ...
  case value-oclexpression then result-oclexpression endcase
  default default-oclexpression enddefault
endswitch
```

Definition 14.1: Switch-case proposed statement

As we have explained, all keywords are mandatory and at least one case section must appear within the switch-case statement.

A common example of use could be as follows:

```
context Person::baseSalary:Real
init: switch (self.category)
            case Category::Director then 5000.0 endcase
            case Category::Engineer then 3500.0 endcase
            case Category::Employee then 1450.0 endcase
            default 1000.0 enddefault
      endswitch
```

Example 14.2: Example of switch-case usage

This example evaluates an attribute of a class that have a type that consists on an enumeration of values. According to the value of the enumeration a different Real value is returned to initialize the attribute *baseSalary* of Person class. If the category does not match with none of the cases, by default a value of 1000.0 is returned.

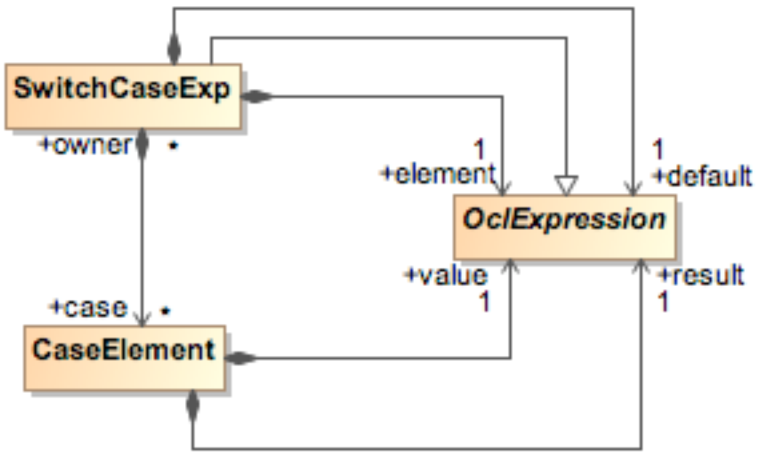


Figure 14.1: Fragment of the OCL 2.0 metamodel including the new switch-case statement

At this image we can see a possible implementation of the OCL metamodel relative part to this imaginary switch-case statement. SwitchCaseExp metaclass is a subclass of OclExpression and contains two instances of it representing the element expression and the expression contained in the default section.

We decided to represent the case sections with a new metaclass called CaseElement that contains an OclExpression specifying the value and another one representing the result to execute if such value matches with the element of the switch-case statement.

In our opinion, this implementation could be a realistic metamodel structure in case of OMG decides to add the switch-case statement into the set of OCL 2.0 expressions.

14.3.2 GRAMMAR CHANGES

Once we have understood how the new statement works, next step consists on to modify our grammar in order to provide the new functionality.

First of all is to add the new tokens into the Token section. In our case we will introduce six new tokens due to token then was added with the if expressions case. It is important to note here that we puts an exclamation mark before the token name to indicate that such token will not be computed. If we do not use this mark, when SableCC-Ext generates the skeleton traversal class the *computeAstFor* methods (remember such methods when we explained SableCC-Ext at section 12.4 of chapter 12) will contain a String with the token name for each token inside the current production.

Once we have introduced the new tokens, we have to include the new productions that will construct the switch-case expression. Production *switch_case_exp_cs* conforms the main rule that specifies how this statement is made.

As we can see, the default section within the switch-case construction is mandatory, although in our case will only be executed (if our parser one day executes OCL expression) when none of the case values matches with the main expression value.

Next to it we have three productions that conforms the list of cases inside a switch expression. This is a common list with at least one case element. It is important to note here that the return object of a *case_list_element_cs* is an *OclCaseBranchExpression* object. This class will have to be implemented because is a utility class that will contain both the value and result expression inside a case section. Therefore, such list of elements will be a list of *OclCaseBranchExpression* objects.

OclCaseBranchExpression class will be like class shown at Figure 14.2. Remember that such kind of classes is placed inside package *astlib* of our structure.

```
Tokens
! switch                = 'switch';
! default               = 'default';
! enddefault            = 'enddefault';
! endswitch             = 'endswitch';
! case                  = 'case';
! endcase               = 'endcase';

Productions
switch_case_exp_cs <SwitchCaseExp> =
   switch paren_open [element]:ocl_expression_cs paren_close
   case_list_cs
   default [defexp]:ocl_expression_cs enddefault
   endswitch
   ;

case_list_cs <List> =
   [element]:case_list_element_cs [tail]:case_list_tail_cs?
   #nocreate
   ;

case_list_element_cs <OclCaseBranchExpression> =
    {case} case [value]:ocl_expression_cs then
```

```
          [result]:ocl_expression_cs endcase
    ;

case_list_tail_cs <List> =
    [tail]:case_list_cs #chain
    ;

primary_exp_cs <OclExpression> =
     {literal} literal_exp_cs #chain
    |{parenthesized} paren_open expression paren_close #chain
    | {property} [prop_call]:property_call_exp_cs #customheritage
                 #nocreate
    | {if} if_exp_cs #chain
    | {switch} switch_case_exp_cs #chain
    ;
```

Definition 14.2: Changes made to OCL 2.0 grammar to support switch-case statement

Finally we do not have to forget to include a new alternative inside production *primary_exp_cs* in order to link with *switch_case_exp_cs* production.

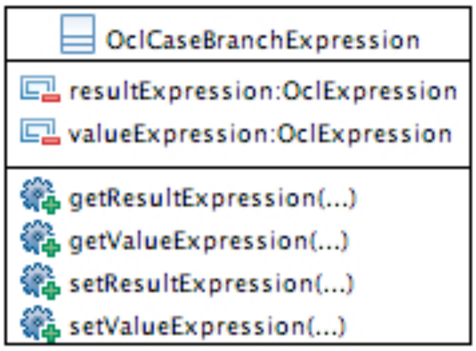


Figure 14.2: OclCaseBranchExpression class

14.3.3 Generated Classes

Once we have introduced the new changes made in our SableCC-Ext specification file, i.e., our OCL 2.0 grammar, next step is to invoke SableCC-Ext in order to obtain the new generated classes including the new skeleton of tree-walker class that will be extended with the switch-case processing.

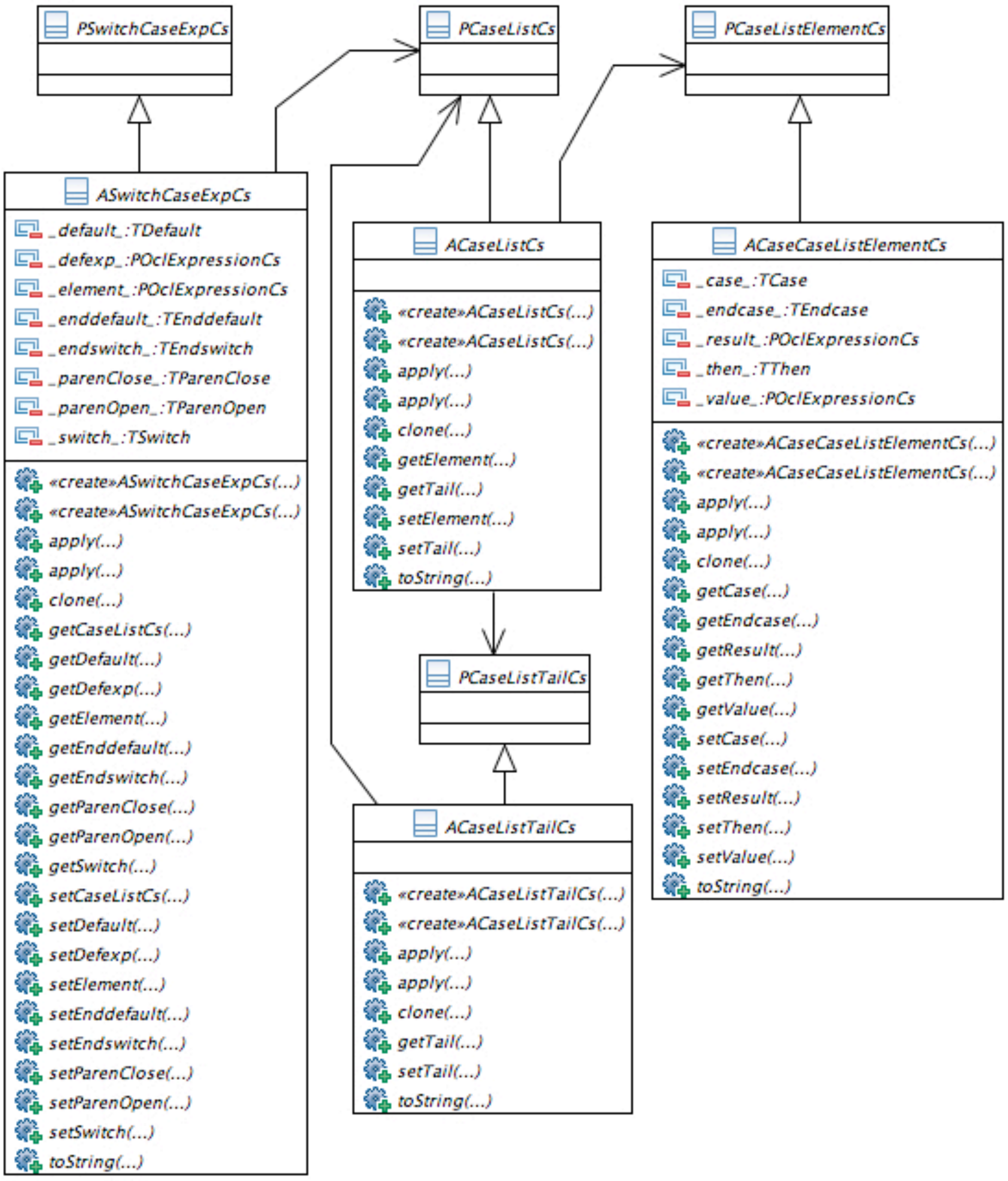


Figure 14.3: Complete SableCC-Ext generated classes for switch-case productions

As you can see in the previous image, each alternative has its own apply method which is essential part of the visitor pattern.

14.3.4 Generated traversal

As explained when we described SableCC-Ext, a skeleton class with the traversal around the grammar productions is provided. Such class is LAttrEvalAdapter and our aim is to extend it to implement the methods that compute AST nodes and that calculate Heritage parameters.

```
public final SwitchCaseExp caseASwitchCaseExpCs(ASwitchCaseExpCs
                  node, Object param) throws AttrEvalException {
  Heritage nodeHrtg = (Heritage) param;
  Heritage childHrtg = null;
  // Descending into element OclExpression to obtain its AST
  POclExpressionCs childElement = node.getElement();
  OclExpression astElement = null;
  if( childElement != null) {
    astElement = (OclExpression) childElement.apply(this,
                  nodeHrtg.copy());
  }
  // Descending into CaseList production to obtain its AST
  PCaseListCs childCaseListCs = node.getCaseListCs();
  List astCaseListCs = null;
  if( childCaseListCs != null) {
    astCaseListCs = (List) childCaseListCs.apply(this,
                            nodeHrtg.copy());
  }
  // Descending into default OclExpression to obtain its AST
  POclExpressionCs childDefexp = node.getDefexp();
  OclExpression astDefexp = null;
  if( childDefexp != null) {
    astDefexp = (OclExpression) childDefexp.apply(this,
                 nodeHrtg.copy());
  }

  // create AST node through factory for current CST node here.
  SwitchCaseExp myAst = (SwitchCaseExp)
                        factory.createNode("SwitchCaseExp");
  // Method to implement. Previous ASTs are parameters here
  myAst = computeAstFor_ASwitchCaseExpCs(myAst, nodeHrtg,
                   astElement, astCaseListCs, astDefexp);
  return myAst;
}

public final List caseACaseListCs(ACaseListCs node, Object param)
                                       throws AttrEvalException {
   Heritage nodeHrtg = (Heritage) param;
```

```
    Heritage childHrtg = null;
    // Descending into an element of the case list
    PCaseListElementCs childElement = node.getElement();
    OclCaseBranchExpression astElement = null;
    if( childElement != null) {
      astElement = (OclCaseBranchExpression)
                   childElement.apply(this, nodeHrtg.copy());
    }
    // Descending into the next elements of the case list
    PCaseListTailCs childTail = node.getTail();
    List astTail = null;
    if( childTail != null) {
       astTail = (List) childTail.apply(this, nodeHrtg.copy());
    }

    // Method to implement. Previous ASTs are parameters here
    List myAst = computeAstFor_ACaseListCs(nodeHrtg, astElement,
                 astTail);
    return myAst;
}

public final OclCaseBranchExpression
       caseACaseCaseListElementCs(ACaseCaseListElementCs
       node, Object param) throws AttrEvalException {

    Heritage nodeHrtg = (Heritage) param;
    Heritage childHrtg = null;
    // Descending into value OclExpression
    POclExpressionCs childValue = node.getValue();
    OclExpression astValue = null;
    if( childValue != null) {
      astValue = (OclExpression) childValue.apply(this,
                 nodeHrtg.copy());
    }
    // Descending into result OclExpression
    POclExpressionCs childResult = node.getResult();
    OclExpression astResult = null;
    if( childResult != null) {
      astResult = (OclExpression) childResult.apply(this,
                  nodeHrtg.copy());
    }
    // create AST node through factory for current CST node here.
    OclCaseBranchExpression myAst = (OclCaseBranchExpression)
               factory.createNode("OclCaseBranchExpression");
    // Method to implement. Previous ASTs are parameters here
    myAst = computeAstFor_ACaseCaseListElementCs(myAst, nodeHrtg,
            astValue, astResult);
    return myAst;
}
```

```
public final List caseACaseListTailCs(ACaseListTailCs node,
                          Object param) throws AttrEvalException {

   Heritage nodeHrtg = (Heritage) param;
   Heritage childHrtg = null;
   // Descending into PCaseListCs production
   PCaseListCs childTail = node.getTail();
   List astTail = null;
   if( childTail != null) {
      astTail = (List) childTail.apply(this, nodeHrtg.copy());
   }
   // create AST node for current CST node here.
   List myAst = astTail;
   return myAst;
}
```

Definition 14.3: Generated methods inside LAttrEvalAdapter.java class

As we said before, all this code is automatically generated by SableCC-Ext. At next section we will implement the methods that compute the AST nodes.

14.3.5 Completing the New Methods

In code inside Definition 14.3 we have seen three methods that must be implemented in order to complete the traversal over the elements in our OCL 2.0 grammar. In the next fragment of Java source code owned by *LAttrGMCAstGenerator.java* we found the implementation for these methods.

First one returns an *OclCaseBranchExpression* object. All we must do is to set both OclExpression AST nodes that are the parameters of this method into the *OclCaseBranchExpression* parameter and return it.

This way we can return a synthesized attribute that contains double information: an OclExpression representing the value expression of a case section and another OclExpression with the result expression of the same case section.

```
public OclCaseBranchExpression
computeAstFor_ACaseCaseListElementCs(OclCaseBranchExpression
            myAst, Heritage nodeHrtg,  OclExpression astValue,
            OclExpression astResult) throws AttrEvalException {
    myAst.setValueExpression(astValue);
    myAst.setResultExpression(astResult);
    return myAst;
}

public SwitchCaseExp computeAstFor_ASwitchCaseExpCs(SwitchCaseExp
             myAst, Heritage nodeHrtg, OclExpression astElement,
             List astCaseListCs, OclExpression astDefexp)
                                      throws AttrEvalException {
    myAst.setElement(astElement);
    myAst.setDefault(astDefexp);

    for (Iterator it = astCaseListCs.iterator(); it.hasNext();) {
       OclCaseBranchExpression cbe =
                       (OclCaseBranchExpression) iterator.next();
       CaseElement ce =
                 (CaseElement) factory.createNode("CaseElement");
       ce.setValue(cbe.getValueExpression());
       ce.setResult(cbe.getResultExpression());
       ce.setOwner(myAst);
    }
    tc.typecheck(myAst);
    return myAst;
}

public List computeAstFor_ACaseListCs(Heritage nodeHrtg,
       OclCaseBranchExpression astElement, List astTail)
                             throws AttrEvalException {
    List result = null;
    if ( astTail != null ) {
       astTail.add(astElement);
       result = astTail;
    } else {
       result = (List) factory.createNode("List");
       result.add(astElement);
    }
    return result;
}
```

Definition 14.4: Methods to control the new switch-case statement behaviour, which are placed inside LAttrGMCAstGenerator.java class

Second method sets the different expressions into the SwitchCaseExp object. Note that for every member of the list passed as a parameter, which is an *OclCaseBranchExpression* object, we have to create a new instance of CaseElement metaclass with the correct values.

Note that last instruction before the return statement is a call to the *typecheck* method of an object instance of the TypeCheck class (tc, in the code). This method makes the type conformance analysis of each element passed as a parameter. As we will in next chapters, TypeCheck class is based in another kind of visitor pattern. An important restriction here is to check that the element expression and each value expression of every case section have compatible types.

Last method only constructs a resultant list joining an element and another list. This method is the one in charge to construct the final list of *OclCaseBranchExpression* elements.

14.3.6 TESTING THE NEW STATEMENT

Finally, once we have finished the implementation and all these steps it is the moment to test if the behaviour of the parser is correct. We have to do the next verifications:

1. Check that the switch-case expression could be written and syntactically checked without errors.

2. Check this new statement with examples that do not produce errors and verify that parser's behaviour is correct.

3. Check the switch-case expression with example that produce errors and verify that such errors are shown to user in a correct way.

4. Verify that the correct type is returned for several switch-case examples of usage.

Once we have done all those kinds of test, we are able to affirm that the processor of OCL expressions supports the switch-case statement.

THE OCL 2.0 PARSER 15

15 THE OCL 2.0 PARSER

15.1 INTRODUCTION

In previous chapters we have studied different OCL tools to be the basis of our project until we decided the best tool to adapt was the Parser Subsystem of the Dresden OCL2 Toolkit. After that, we studied the main components and utilities of such tool and we showed the specification file containing the OCL 2.0 grammar used to generate the framework for to implement our processor.

At this chapter we finally introduce the processor of OCL 2.0 expression, also known as OCL 2.0 parser here.

First of all to show the complete structure of the parser is mandatory to understand its different components and how are they related. Then, we will expose a simple example of usage that will be useful to be familiar with our tool.

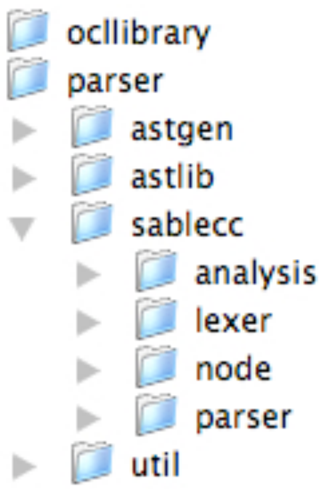
Finally we show the conversion process made by the OCL 2.0 parser. For every expression in the OCL 2.0 language we show all the conversion steps from the textual constraint to the XMI representation of the metamodel instances of such expression. This section also contains the explanation about the inverse conversion from metamodel instances to textual constraints, and indications of how to delete processed constraints.

Therefore, we could affirm that it is the main chapter of this document due to it is the union point between all concepts explained in the other chapters.

15.2 STRUCTURE

Finally, our processor of OCL 2.0 expressions has been included inside the release package of the Eina GMC conceptual modeling environment. Inside the source package of such tool we can found the parser directory, which contains all Java classes of the processor. Additionally, package ocllibrary contains important information to explain.

Here we will explain in detail the structure of these directories and its subdirectories in order to understand the different components and to know how are placed in the hierarchy of packages.



```
ocllibrary
parser
  ▶ astgen
  ▶ astlib
  ▼ sablecc
      ▶ analysis
      ▶ lexer
      ▶ node
      ▶ parser
  ▶ util
```

Figure 15.1: Hierarchy of packages of our processor

15.2.1 PACKAGE ASTGEN

Inside this package we find the most important Java classes of the processor of OCL 2.0 expressions. At next subsections such classes will be explained.

15.2.1.1 LAttrGMCAstGenerator.java

Such class, as we explained earlier, contains the implementation of the methods that deal with the checking and instantiation of the elements inside OCL expressions. LAttrGMCAstGenerator is subclass of LAttrEvalAdapter, which is the generated attribute evaluator skeleton automatically generated by SableCC-Ext.

We can affirm that this class is the brain of our processor because includes the instructions to make the instantiation inside the UML and OCL metamodels and the calls to the type checker component. If we need to change the behaviour of our processor we must change the methods of this file.

15.2.1.2 NodeFactory.java

Another important element inside such package is the Node Factory component that is written inside the NodeFactory.java file. Such element was explained at previous chapter. As we said then, it is based in a HashMap of pairs consisting in the name of an element and the methods to create it. Calling the *createNode* method with a String as a parameter specifying the name of the object to create we obtain a new object of such type.

In order to extend this factory of nodes we only have to add a new pair similar than existent ones into the core HashMap and automatically we will be able to use it.

15.2.1.3 Heritage.java

It is important to remember that in the parsing process there are two kinds of information shared between nodes. The information passed from parents to its descendants is known as inherited attributes and to simulate this behaviour we have the Heritage.java class. Instances of such class store information related to a parent node in order to pass it to its descendants or siblings inside the tree traversal of the compilation process.

This class consists basically in a set of attributes with their setters and getters methods to access them. It is very similar than Dresden implementation but we have deleted some attributes that are not needed in our case and we also have included some new ones.

Among others, Heritage stores information about the contextual class of a constraint, the type of expressions in where the nodes are placed (e.g., iterators, postconditions, ...), or also the source expression of the current one.

In order to extend such class we only have to add new attributes with their own setter and getter methods to be accessed.

15.2.1.4 Environment

The Environment in our traversal is an attribute included within the Heritage object. It contains information about the accessible elements in a current moment of the parsing process.

For example, inside a body expression of an iterator with iterator variables previously declared, we can use such variables to access the elements of the source expression (that should be a collection). Inside the checking of this body expression, such variables are not directly accessible because they were processed before. Therefore, to check inside a body expression if a variable is correctly used because it was defined in the iterator variables' zone, we only have to search in the current environment if such variable exists.

At the beginning of a constraint parsing process we always introduce a new variable inside the environment representing the *self* element due to be used within the expressions inside such constraint.

Each environment can contain a reference to another subenvironment in order to provide the possibility of emulate a symbol table, which is a common concept in compilation that specifies different levels of storage. Therefore, if there are two elements inside the environment with the same name, when we search by such name we will find the element that is closer to the environment in use. In the same way, if we find an element that is not stored in the current environment, the searching process will continue by the parent environments.

15.2.2 Package astlib

Inside this package there are placed the utility classes which names begin with *Ocl* and are used to simplify sharing information between different productions of our grammar.

These classes are inherited from the Dresden implementation of their Parser Subsystem and we only made some little changes to adapt it to the Eina GMC conceptual modeling environment, although they are very similar than originals.

They are used in the OCL 2.0 grammar for the SableCC-Ext in order to wrap some elements to be passed as intermediate synthesized attributes to another parent productions that will use them to create the final synthesized attributes. Inside the productions of such grammar we can find their names between angle brackets indicating that the result object of the production is an instance of one of these classes.

15.2.3 PACKAGE SABLECC

This package contains all generated classes by SableCC-Ext, as we explained at previous chapters. It is divided on four subpackages that will be explained here.

15.2.3.1 Package analysis

The *analysis* package contains the classes introduced by SableCC and SableCC-Ext to make the attribute evaluation and the parse tree traversal. Such classes are:

- Analysis.java: basic interface generated by SableCC to specify the methods of a visitor class.
- AnalysisAdapter.java: basic implementation of the Analysis.java interface to support the visitor pattern traversal.
- AnalysisWithReturn.java: version of the Analysis.java interface providing support for to return parameters and to pass objects as parameters in a extended visitor architecture.
- DepthFirstAdapter.java: subclass of AnalysisAdapter.java in which the visitor traversal is done following a depth-first strategy.
- ReversedDepthFirstAdapter.java: subclass of AnalysisAdapter.java in which the visitor traversal is done following a reversed depth-first strategy.
- LAttrEvalAdapter.java: automatically generated attribute evaluator skeleton by SableCC-Ext. As we explained in previous chapters, makes the parse tree traversal and makes easier the evaluation because we only have to implement the empty methods in order to complete the whole treatment for the elements of the parse tree.

- AttrEvalException.java: Java exception to thrown in case of error in the attribute evaluation process.

It is important to note here that the most important class here is the generated skeleton of the attribute evaluator because is the core of our processor. Changes on the grammar will directly appear as changes on such class.

15.2.3.2 Package lexer

This package contains both the Lexer.java class and the LexerException.java. First one is the element that tokenizes the input of our processor according to the token definition of the specification file for SableCC-Ext that we explained at chapter 13.

The result of the execution of this class is a stream of tokens representing the processed input. Furthermore, if any error occurs in this process the LexerException is thrown in order to alert the users.

15.2.3.3 Package node

This package contains all generated nodes by SableCC-Ext according to the specification file of our OCL 2.0 grammar. At chapter 13 where we explained our grammar we studied the different productions and alternatives, and then we showed several diagrams with these generated classes.

It is important to emphasize that generated classes representing productions are abstract Java classes and their names begin with a 'P' character. On the other hand, generated classes representing alternatives within a production are concrete Java classes extending abstract ones and their names begin with an 'A' character. Finally, classes representing tokens are also concrete subclasses of Token.java class and their names begin with a 'T' character.

15.2.3.4 Package parser

This package contains the Parser.java class representing the Parser component of our processor of OCL 2.0 expressions. The result of to apply the *parse* method of such class is the first node of the parse tree, represented by an instance of the *Start.java* class from node package.

If any error is found during the parsing process, an instance of the ParserException.java class is thrown in order to alert users.

Inside this package we also find an extra class called TokenIndex.java that identifies every token of the Token section of our specification file for SableCC-Ext with an integer number. Such identifier is internally used by different components along our parsing process.

15.2.4 PACKAGE UTIL

This is the last package placed inside the package parser. As we will see, inside this directory we find Java classes used to simplify some processes of our processor.

15.2.4.1 Utility.java

This class contains a lot of methods used inside several components, like the AST Generator LAttrGMCAstGenerator.java.

Among others, Utility class contains methods to verify if exists connection between to association ends, to check if a property is owned by an UML class or to create new OCL metamodel instances from others previously created.

All methods without a specific place were allocated inside this class. Therefore here we can find a lot of different kinds of methods.

15.2.4.2 ReflectiveVisitor.java

This is a very important class that works with the Java Reflective API [JRAw] in order to make a special visitor that discovers the class type of the objects accessing to their Java internal description.

As explained at previous chapter, we only have to write a new class extending this one where a visitor method is implemented for each element to process. Such methods have a parameter with the same class than we have to compute within it.

Then, there exists a general method to be called for each element. Such method uses the Reflective API to verify the class type of the object parameter and finally searches inside the new Java class extending ReflectiveVisitor.java in order to find the method that has a parameter of such class type. Once it has found the method, invokes it with the object to compute as parameter.

At next section TypeCheck.java will be studied, as an implementation of the ReflectiveVisitor.

15.2.4.3 TypeCheck.java

TypeCheck class is the component that does the type checking of the elements of the OCL metamodel. It implements the ReflectiveVisitor explained before. Inside this class we found a general method named *typecheck* that admits one parameter. Such parameter is the object to need a check for its types.

For every element to check there exists an *evaluate* method with one parameter. The parameter of such method identifies the element to process. To understand this behaviour we will see a simple example of usage with an element instance of IfExp metaclass representing an *if* expression.

There exists a general method to be called for every element to process:

```
public Type typecheck (TypedElement te)
          throws WellFormednessException , AttrEvalException
```

Such method can be called for an IfExp element:

```
// imagine that tc is an instance of TypeCheck class
// and ifexp an instance of IfExp metaclass
tc.typecheck(ifexp);
```

This invocation implies that Reflective Visitor searches and then has to find a method with the next signature:

```
public void evaluate(IfExp exp) throws AttrEvalException
```

This method is called for the ifexp instance of IfExp metaclass. Inside this method we could do as follows:

```
OclExpression cond = exp.getCondition();
OclExpression thenBranch = exp.getThenExpression();
OclExpression elseBranch = exp.getElseExpression();
Type condType = typecheck(cond);
Type thenType = typecheck(thenBranch);
Type elseType = typecheck(elseBranch);
```

Using this kind of visitor we only have to obtain the attributes of IfExp metaclass and then call the typecheck method of each one to obtain their types. This recursive strategy simplifies the type checking process.

On this example, last step consists in to calculate the type of the IfExp from the types of its attributes.

Example 15.1: Usage of the TypeCheck class as a subclass of the ReflectiveVisitor

15.2.5 PACKAGE OCLLIBRARY

Inside this package we find the OclLibrary.java file that contains and implementation of a factory of operations. For each operation specified in the OCL 2.0 specification document [Obj06] we have created an entry in a HashMap. As we explained at previous chapter related to the NodeFactory component, the OclLibrary maintains the same structure. We implemented a generic method that needs a String as parameter. Such String is used to indicate the name of the OCL 2.0 operation to find.

This method is called findOclOperationByName and first of all verifies if an operation with the searching name exists in the metadata repository. If affirmative returns it but if the operation is not found, it is created and then returned.

With this design decision we avoid to have repeated operations inside the metadata repository.

Furthermore, inside this Java class we have methods to do the same as with operations but with the basic types described inside the OCL 2.0 specification. These types are String, Boolean, Real and Integer.

It is important to emphasize that separating the creation of types and operations of the OCL 2.0 Library from the main code provides an increasing changeability that can be useful if the OCL language is extended or modified.

15.2.6 Package examples

Finally, there exists another secondary package named *examples* that obviously contains some classes implementing different examples of usage. The main example here is the DemoOcl20Parser.java file that contains the source code that constructs the Demo program presented at Figure 12.6.

15.3 HOW TO USE THE PARSER

15.3.1 Usage example

At this section we will explain the steps that user must follow in order to work with the processor of OCL 2.0 expressions.

15.3.1.1 How to obtain an XMI file

To begin our job, it is mandatory to own an XMI file that specifies a correct UML conceptual model.

At chapter where we introduced Eina GMC (see chapter 5, section 5.5) as our conceptual modeling environment it is possible to find more information about how to obtain an XMI file from a CASE tool (in our case, Poseidon for UML), or directly instantiating model elements writing the needed Java code. Here we will suppose that you know the way of obtaining XMI files.

It is important to emphasize that a precondition for our processor of OCL 2.0 expressions is that every element inside the XMI file representing an UML model that can have a type has such type specified inside such XMI file. For example, the type of the attributes inside an UML class must not be empty. Otherwise our processor cannot work properly due to its type conformance checking process will fail because it will not find the types to check.

15.3.1.2 Creating a new project

In order to prepare the environment of our processor, we have to create an instance of the Eina GMC class Project.

```
Project p; String name = “myProject”;
p = new Project(name);
System.out.println(“Project “+name+” has been created”);
```

Example 15.2: Creating a new project

We can opt to set a name to such Project instance. Once we have done it, it is the moment of loading the XMI file containing the representation of an UML model.

```
String XMIfileName = "/Users/Desktop/XMI/mymodel.xmi";
p.importXMI(XMIfileName);
System.out.println("Import finished");
```

Example 15.3: Importing an XMI file

With these easy steps we have the project created and the information contained inside the XMI file loaded within the metadata repository of the Eina GMC conceptual modeling environment.

It is important to note that Eina GMC library must be imported here in order to avoid errors of binary files not found. Such library can be found in JAR format at Eina GMC web site [GMCw]. The binary Java classes of our processor are also included in such library.

15.3.1.3 Creating the components of our parser

Next step after we have both project and model loaded inside the metadata repository consists on to construct the components of our parser.

```
// text variable contains OCL 2.0 constraints
String text = "context Person inv: ...
               context Project inv: ...
               ...
               context Work::salary:Real init: ... ";
Lexer lexer = new Lexer (
                  new PushBackReader(
                      new StringReader(text),1024));
Parser parser = new Parser(lexer);
```

Example 15.4: Creation of lexer and parser components

As we can see, first of all we use a String variable to store the OCL 2.0 textual constraints. With such String we must create a Java PushBackReader object in order to pass it to the constructor method of the lexical analyzer (lexer).

Finally, to construct the parser component we have to use the previous lexer as a parameter of its constructor method.

At this point lexical and syntactical errors are returned as Java Exceptions as we will explain later.

15.3.1.4 Processing OCL expressions

To process OCL constraints is a very simple task if we have followed the previous steps.

```
Start ast = parser.parse();
ast.apply(new LAttrGMCAstGenerator(p), new Heritage());
```

Example 15.5: Processing the compilation tree

We have to call the *parse()* method of the Parser object, which will return us the AST of the processed constraints. Concretely we will obtain a reference to the first node of such tree, which is an instance of the Start class.

Next step consists on to use the visitor pattern in order to begin the tree traversal through all tree nodes. We can do it calling the *apply* method of the previous reference, with two parameters.

First parameter is an instance of LAttrGMCAstGenerator class, which needs the initial Project object of the beginning as a parameter for its constructor method. This new instance deals with the necessary treatment for each node of the compilation tree, creating the correspondent instances into the OCL metamodel of the metadata repository.

Second parameter is a new instance of Heritage class, which deals with sharing information between parent nodes and its descendants.

In case of existing problems on this process, each error is returned as a Java exception.

15.3.1.5 Store results into an XMI file

Once we have processed the OCL expressions, next step is to save the changes made within the metadata repository (new instantiation of OCL expressions as OCL metamodel instances) in a new XMI file.

```
String
outputXMIfile(“/Users/Desktop/mymodelwithconstraints.xmi”);
p.saveXMI(outputXMIfile);
```

Example 15.6: Storing metadata into a new XMI file

It is important to close the current instance of Project calling the *closeProject()* method once we have finished to work with it in order to avoid inconsistence problems with the metadata repository of the Eina EMG environment.

```
p.closeProject();
```

Example 15.7: Method to close the current project and its metadata repository

15.3.2 Error Feedback

In our processor of OCL 2.0 expressions, every time that an error is found a Java exception is thrown with the information of the error and where it occurs.

There are different kinds of exceptions depending on the stage of the process in which the error happens. We will show these kinds in the next list:

- LexerException: exception thrown if an error happens during the lexical analysis of the input constraints.

- ParserException: exception thrown if an error occurs during the execution of the *parse()* method of Parser class.

- AttrEvalException: exception thrown if an error occurs during the evaluation of attributes and the instantiation into the metamodel of the OCL input constraints.

- WellFormednessException: this exception is subclass of AttrEvalException and it is used to be thrown when an error is found during the type conformance checking process.

15.3.3 Rollback

Actually, Eina GMC uses the MDR[MDRw] metadata repository provided by Netbeans community[NETw]. Such repository allows using transactions like in databases.

This feature is useful in case of need a rollback in the changes made inside the repository, if finally we have to abort the processing of OCL expressions due to semantic errors (like an incompatibility of types in an arithmetical expression).

We will explain here how to use this feature inside the previous code that constructs the different components of our processor of OCL 2.0 expressions in order to avoid an inconsistent metadata repository due to incomplete constraints.

```
// get the default repository
MDRepository repository =
             MDRManager.getDefault().getDefaultRepository();
// start a write transaction
repository.beginTrans(true);
try {
   // Code where the repository is modified
   Start ast = parser.parse();
   ast.apply(new LAttrGMCAstGenerator(p), new Heritage());
} catch (Exception e){
   // We can do something with the exception
   e.printStackTrace();
} finally {
   // release the transaction
   repository.endTrans(true);
}
```

Example 15.8: Basic usage of rollback in transactions

We can observe that first we obtain the repository reference that is being used actually, in order to be able to call the *beginTrans* and *endTrans* methods with the try/catch Java structure.

First parameter that we pass to *beginTrans* specifies if the current transaction is a write transaction (true) or a read-only transaction (false). In order to allow rollback (erase all changes made to achieve to be in an earlier initial state) we have to pass a true Boolean value.

Such mechanism of rollback is executed when an exception is captured inside the try construction. Therefore, when an error is found during the parsing process an exception is thrown and causes that all new instances created within such process inside the metadata repository are deleted in order to recover the initial state of such repository.

15.3.4 FACADE OPERATIONS

There exists another way of process OCL 2.0 constraints. In order to simplify the usage of our processor we provide some operations inside a facade class that directly constructs the processor's components.

Inside the Java class *ParserFacade* placed into the package facadeOCL we find the *addOclExpression* method that has two parameters. First one is a String representing the OCL 2.0 constraints to parser. Second parameter consists in one instance of *Project* that must have imported a UML model previously.

15.4 CONVERSION PROCESS

Along this section we will introduce the conversion that our processor of OCL 2.0 expressions is able to do. First of all a diagram is shown containing the instantiation of each element or construction of the OCL language. Then, we will explain the XMI fragment generated for such diagrams.

15.4.1 CONSTRAINT CONTAINERS

As we explained in previous chapters, the OCL 2.0 language provides a set of constraint containers, also known as context structures. Here we will show the conversion process of these constructions as wrappers for the other OCL expressions.

15.4.1.1 Invariants and definitions

Invariants and definitions are constraint containers that have a Classifier as a context. At next example we will remember the templates for to write both invariants and definitions.

Invariant:
```
context Class
inv constraintName : <insert invariant expression here>
```
Definition of attribute:
```
context Class
def constraintName : newAttribute:Type
                     = <insert definition expression here>
```
Definition of query operation:
```
context Class
def constraintName : newQueryOperation(parameter list):Type
                     = <insert query expression here>
```

Example 15.9: Invariants and definitions construction templates

As we have seen, a definition context can define a new attribute or a new query operation. Both invariants and definitions share the same structure of instantiation, as we can see at below image.

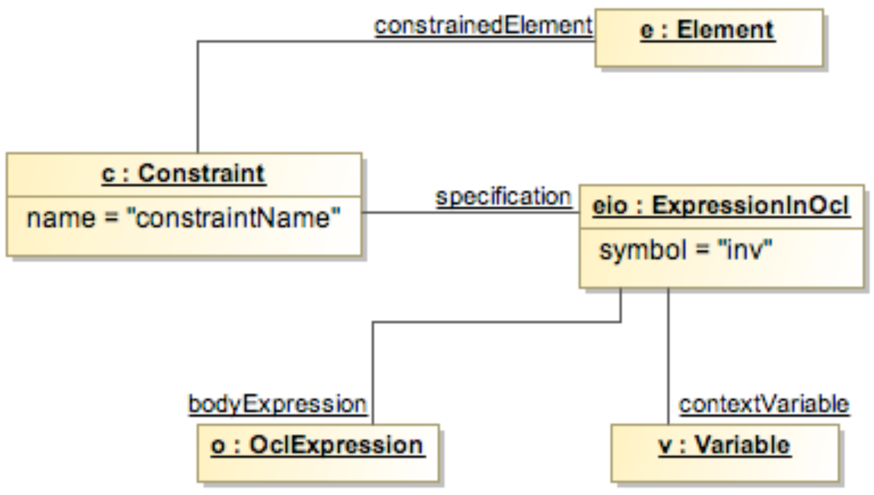


Figure 15.2: Instantiation for an invariant context construction

At Figure 15.2 we show the instantiation of an invariant. It is important to note here that the only change between the instantiation of an invariant and the instantiation of a definition is the value of *symbol* attribute owned by the ExpressionInOcl instance.

Due to Eina GMC does not support stereotypes we decided to identify the kind of a constraint using such attribute of this instance. Therefore, when we are processing an invariant the *symbol* attribute will have the 'inv' value whereas when we process a definition it will have the 'def' value.

When Eina GMC supports stereotypes we will opt to use them to apply an <<invariant>> or <<definition>> stereotype to the constraint instance.

```
<uml2.Kernel.Constraint xmi.id = 'id1' name = 'constraintName'
 visibility = 'public'>
   <uml2.Kernel.Constraint.constrainedElement>
         <uml2.Kernel.Class xmi.idref = 'classRef'/>
   </uml2.Kernel.Constraint.constrainedElement>
   <uml2.Kernel.Constraint.specification>
      <uml:ExpressionInOcl xmi.id = 'id2' symbol = 'inv'>
         <uml:ExpressionInOcl.bodyExpression>
            <uml:OperationCallExp xmi.idref = 'opRef'/>
         </uml:ExpressionInOcl.bodyExpression>
         <uml:ExpressionInOcl.contextVariable>
            <uml:Variable xmi.idref = 'var1'/>
         </uml:ExpressionInOcl.contextVariable>
      </uml:ExpressionInOcl>
   </uml2.Kernel.Constraint.specification>
</uml2.Kernel.Constraint>

<uml:Variable xmi.id = 'var1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Class xmi.idref = 'classRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:Variable.selfOwner>
      <uml:ExpressionInOcl xmi.idref = 'id2'/>
   </uml:Variable.selfOwner>
</uml:Variable>
```

Definition 15.1: XMI representation for an invariant context construction

Finally, at previous definition we can see the conversion of the instantiation for an invariant constraint. Note that the name of the constraint is stored in the header tag of the Constraint element. In this case, the constrained element is a class because the context is an invariant. In case of a definition, such constrained element could be the property or the operation defined.

The body expression of the invariant conversion in this case is an OperationCallExp although it could be another instance of OclExpression. Furthermore, the context variable is the same in both cases and references the class where the constraint is placed. We can see the XMI representation of such variable whose type is the class that we have mentioned.

15.4.1.2 Initializations and derivations

Another kinds of constraints are the initializations and derivations constructions. They can be written as explained in the example below.

Initialization:
```
context Class::property:Type
init constraintName: <insert initialization expression here>
```
Derivation:
```
context Class::property:Type
derive constraintName: = <insert derivation expression here>
```

Example 15.10: Initializations and derivations construction templates

It is important to remember here that a property can represent both an attribute and an association end.

Initializations and derivations are constructions that share the same template. In this case, the constrained element of these constraints is a property. The only difference from the instantiation of an invariant or definition with respect to initializations and derivations is the symbol of the ExpressionInOcl instance that identifies the kind of the constraint. In this case, we use ‘init’ for initializations and ‘derive’ for derivations.

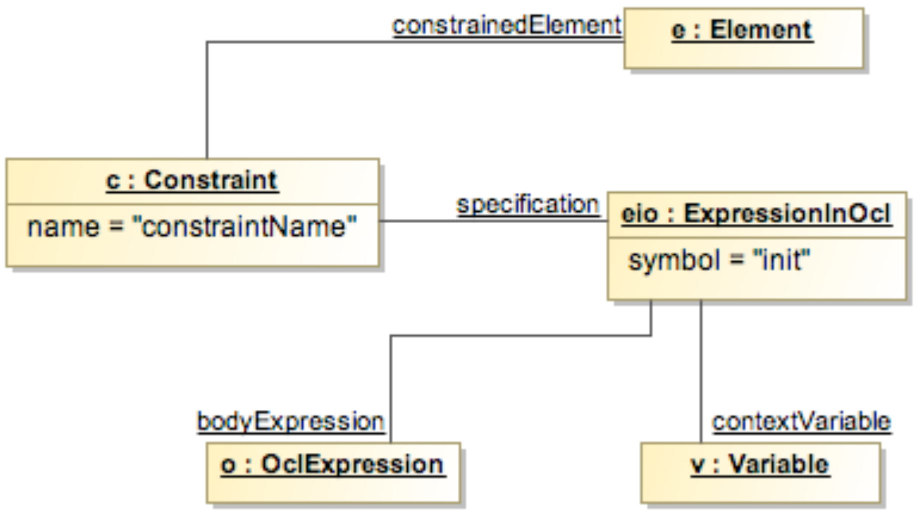


Figure 15.3: Instantiation for an initialization context construction

As we can see, both diagram and XMI representation are very similar than invariants and definitions.

```
<uml2.Kernel.Constraint xmi.id = 'id1' name = 'constraintName'
 visibility = 'public'>
   <uml2.Kernel.Constraint.constrainedElement>
      <uml2.Kernel.Property xmi.idref = 'propertyRef'/>
   </uml2.Kernel.Constraint.constrainedElement>
   <uml2.Kernel.Constraint.specification>
      <uml:ExpressionInOcl xmi.id = 'id2' symbol = 'init'>
         <uml:ExpressionInOcl.bodyExpression>
            <uml:IntegerLiteralExp xmi.idref = 'ileRef'/>
         </uml:ExpressionInOcl.bodyExpression>
         <uml:ExpressionInOcl.contextVariable>
            <uml:Variable xmi.idref = 'var1'/>
         </uml:ExpressionInOcl.contextVariable>
      </uml:ExpressionInOcl>
   </uml2.Kernel.Constraint.specification>
</uml2.Kernel.Constraint>

<uml:Variable xmi.id = 'var1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Class xmi.idref = 'classRef'/>
```

```
    </uml2.Kernel.TypedElement.type>
    <uml:Variable.selfOwner>
       <uml:ExpressionInOcl xmi.idref = 'id2'/>
    </uml:Variable.selfOwner>
 </uml:Variable>
```

Definition 15.2: XMI representation for an initialization context construction

15.4.1.3 Preconditions, postconditions and body of operations

Finally, the last kind of constraints is the operation contexts. Concretely they are preconditions, postconditions and body of operations.

We can remember their syntax looking at the next examples.

Precondition:
```
context Class::operation(list of parameters):Type
pre constraintName: <insert precondition expression here>
```
Body of operation:
```
context Class::operation(list of parameters):Type
body constraintName: = <insert body expression here>
```
Postcondition:
```
context Class::operation(list of parameters):Type
post constraintName: <insert postcondition expression here>
```

Example 15.11: Preconditions, postconditions and body of operations construction templates

Such constructions have the same metamodel instantiation than previous ones, but in this case we use 'pre', 'post' or 'body' as values for the symbol attribute of ExpressionInOcl instance. Furthermore, the constrained element here is always an operation of the conceptual model.

Now we will introduce the instantiation diagram and the XMI representing such instantiation for a common precondition.

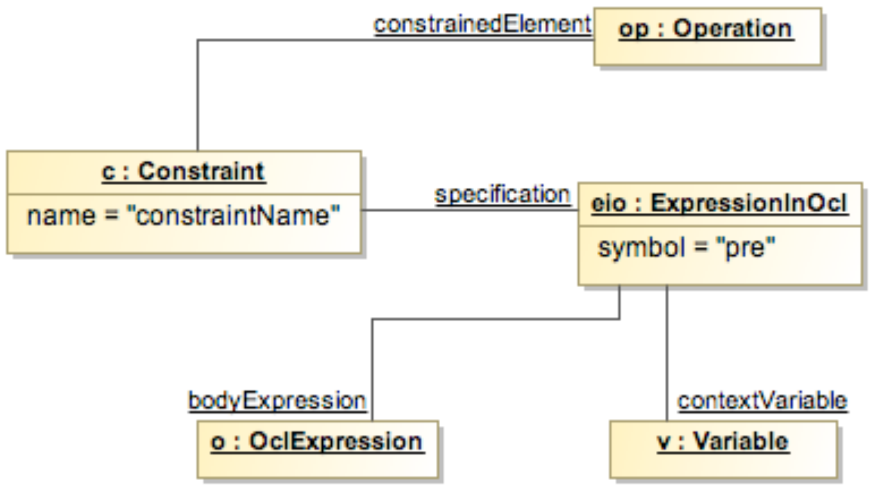


Figure 15.4: Instantiation for a precondition construction

```
<uml2.Kernel.Constraint xmi.id = 'id1' name = 'constraintName'
 visibility = 'public'>
   <uml2.Kernel.Constraint.constrainedElement>
      <uml2.Kernel.Operation xmi.idref = 'operationRef'/>
   </uml2.Kernel.Constraint.constrainedElement>
   <uml2.Kernel.Constraint.specification>
      <uml:ExpressionInOcl xmi.id = 'id2' symbol = 'pre'>
         <uml:ExpressionInOcl.bodyExpression>
            <uml:BooleanLiteralExp xmi.idref = 'bleRef'/>
         </uml:ExpressionInOcl.bodyExpression>
         <uml:ExpressionInOcl.contextVariable>
            <uml:Variable xmi.idref = 'var1'/>
         </uml:ExpressionInOcl.contextVariable>
      </uml:ExpressionInOcl>
   </uml2.Kernel.Constraint.specification>
</uml2.Kernel.Constraint>

<uml:Variable xmi.id = 'var1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Class xmi.idref = 'classRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:Variable.selfOwner>
      <uml:ExpressionInOcl xmi.idref = 'id2'/>
   </uml:Variable.selfOwner>
</uml:Variable>
```

Definition 15.3: XMI representation for a precondition context construction

15.4.2 Types

In this section we will describe the conversion process for the types used in the OCL 2.0 language.

15.4.2.1 DataTypes

As was explained earlier, in the specification document of the OCL 2.0 language there are four types: Integer, Real, String and Boolean. We use them as instances of DataType metaclass as shown at next image.

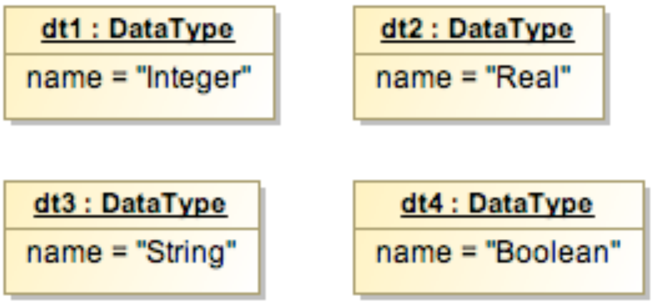


Figure 15.5: Instantiation for DataTypes of basic types

The representation of such data types into the XMI language supported by the Eina GMC conceptual modeling environment is as follows:

```
<uml2.Kernel.DataType xmi.id = 'dt1' name = 'Integer'
  qualifiedName = '' isLeaf = 'false' isAbstract = 'false'>
</uml2.Kernel.DataType>
<uml2.Kernel.DataType xmi.id = 'dt2' name = 'Real'
  qualifiedName = '' isLeaf = 'false' isAbstract = 'false'>
</uml2.Kernel.DataType>
<uml2.Kernel.DataType xmi.id = 'dt3' name = 'String'
  qualifiedName = '' isLeaf = 'false' isAbstract = 'false'>
</uml2.Kernel.DataType>
<uml2.Kernel.DataType xmi.id = 'dt4' name = 'Boolean'
  qualifiedName = '' isLeaf = 'false' isAbstract = 'false'>
</uml2.Kernel.DataType>
```

Definition 15.4: XMI representation for datatypes of basic types

15.4.2.2 CollectionType

There exist four categories of collection types: BagType, SetType, OrderedSetType and SequenceType. Each one contains a reference to the type of its elements. We will show an example of BagType whose elements are Integers: `Bag(Integer)`.

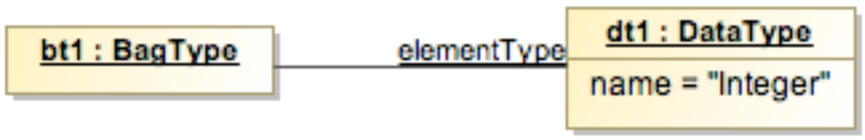


Figure 15.6: Instantiation for a BagType whose element type is Integer

The previous instantiation has a direct conversion. It is important to note that these types also contain a special field inside the XMI representation where all elements with such types are referenced. This field is named *_typedElementOfType* as we can see at next XMI code.

```
<uml:BagType xmi.id = 'bt1' name = 'Bag(Integer)' isLeaf = 'false'
 isAbstract = 'false'>
   <uml2.Kernel.Type._typedElementOfType>
      <uml:CollectionLiteralExp xmi.idref = 'cleRef'/>
   </uml2.Kernel.Type._typedElementOfType>
   <uml:CollectionType.elementType>
      <uml2.Kernel.DataType xmi.idref = 'integerRef'/>
   </uml:CollectionType.elementType>
</uml:BagType>
```

Definition 15.5: XMI representation for a Bag collection type

15.4.2.3 TupleType

We should remember here that a TupleType is a type that specifies a tuple and its members. TupleType has the same structure than a UML class. Therefore its members are instances of Property.

At next image we can see an instantiation example for a TupleType that only has an attribute named 'a': `TupleType(a:Integer)`

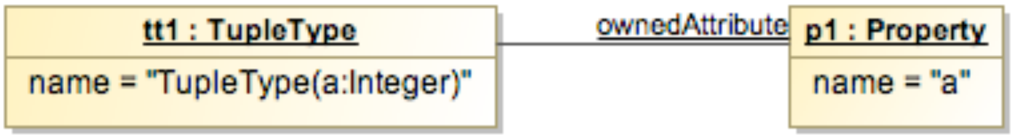


Figure 15.7: Instantiation for a tuple type with only one owned attribute

According to it, the representation of a TupleType with the XMI language is like the code inside next definition.

```
<uml:TupleType xmi.id = 'tt1' name = 'TupleType(a:Integer)' isLeaf
 = 'false' isAbstract = 'false'>
   <uml2.Kernel.Type._typedElementOfType>
      <uml:TupleLiteralExp xmi.idref = 'tleRef'/>
   </uml2.Kernel.Type._typedElementOfType>
   <uml2.Kernel.Class.ownedAttribute>
      <uml2.Kernel.Property xmi.id = 'p1' isUnique = 'false'
      isOrdered = 'false' lower = '0' upper = '0' name = 'a'
      isLeaf = 'false' isStatic = 'false' isReadOnly = 'false'
      isDerived = 'false' isDerivedUnion = 'false' isComposite =
      'false'>
         <uml2.Kernel.TypedElement.type>
            <uml2.Kernel.DataType xmi.idref = 'integerRef'/>
         </uml2.Kernel.TypedElement.type>
         <uml2.Kernel.Property.tupleLiteralPart>
            <uml:TupleLiteralPart xmi.idref = 'tlpRef'/>
         </uml2.Kernel.Property.tupleLiteralPart>
      </uml2.Kernel.Property>
   </uml2.Kernel.Class.ownedAttribute>
</uml:TupleType>
```

Definition 15.6: XMI representation for a tuple type

Inside the main TupleType tag we find a section named *_typedElementOfType* where all elements that have such type are referenced. Furthermore, there is another section including all owned attributes. Such section contains the definition of only one Property here due to in our example we only have one member. Such

Property contains its name, its type (a reference to the DataType Integer), and finally a reference to the TupleLiteralPart that owns it.

15.4.3 LITERAL EXPRESSIONS

At this point we will describe the conversion process of the literal expressions of the OCL 2.0 language.

15.4.3.1 Numeric literals

First literals to study are the numeric literals composed by both RealLiteralExp and IntegerLiteralExp. They are the metaclasses that wrapp Real and Integer constant values used inside OCL expressions.

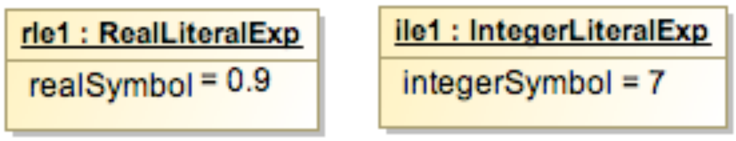


Figure 15.8: Instantiation for Real and Integer literals

It is important to note that in the XMI representation of such classes there exists a link to the type reference of such instance. In our case, links to Real and Integer data types.

```
<uml:RealLiteralExp xmi.id = 'rle1' name = '' realSymbol = '0.9'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'realRef'/>
   </uml2.Kernel.TypedElement.type>
</uml:RealLiteralExp>

<uml:IntegerLiteralExp xmi.id = 'ile1' name = ''
integerSymbol = '7'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerRef'/>
</uml:IntegerLiteralExp>
```

Definition 15.7: XMI representation for numeric literals

15.4.3.2 String literals

Similarly than with numeric literals, to store String constant values we use the StringLiteralExp metaclass.

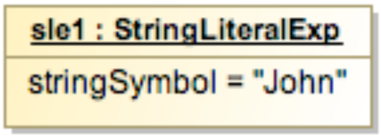


Figure 15.9: Instantiation for an String literal representing the 'John' value

The instantiation for these cases is represented using the XMI language as shown below.

```
<uml:StringLiteralExp xmi.id = 'sle1' name = 'John'
stringSymbol = 'John'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'stringRef'/>
   </uml2.Kernel.TypedElement.type>
</uml:StringLiteralExp>
```

Definition 15.8: XMI representation for String literal

15.4.3.3 Boolean literals

To complete the instantiation of basic literals we have to specify how to convert a Boolean constant value (true or false). To do this we use the BooleanLiteralExp metaclass that stores such value.

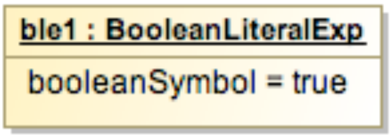


Figure 15.10: Instantiation for a Boolean value

The conversion of it into the XMI language of the Eina GMC is very similar than other basic literal expressions. In this case the type reference links with the Boolean data type.

```
<uml:BooleanLiteralExp xmi.id = 'ble1' name = ''
booleanSymbol = 'true'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'booleanRef'/>
   </uml2.Kernel.TypedElement.type>
</uml:StringLiteralExp>
```

Definition 15.9: XMI representation for Boolean literal

15.4.3.4 Enumeration literals

Another kind of literals is enumeration literal expressions. To use an enumeration literal expression it is necessary to write the Enumeration name followed by a double colon and the enumeration literal. In our example we can see the instantiation of an Enumeration called Gender. It has two literals, male and female.

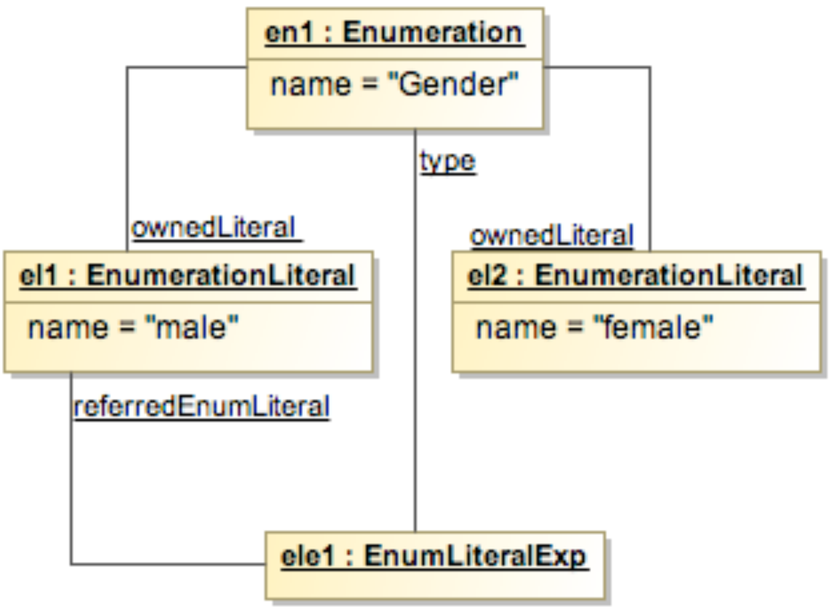


Figure 15.11: Instantiation of an enumeration literal

The previous instantiation shows a EnumLiteralExp representing the `Gender::male` literal expression. Note that the type of such expression is the Enumeration used. Enumeration literal expressions can be used to initialize attributes or in comparisons.

The next XMI code contains firstly the instantiation of the previous enumeration according to the UML metamodel, and then the representation of the EnumLiteralExp according to the OCL metamodel.

```
<uml2.Kernel.Enumeration xmi.id = 'en1' name = 'Gender'
isLeaf = 'false' isAbstract = 'false'>
   <uml2.Kernel.Type._typedElementOfType>
      <uml2.Kernel.Property xmi.idref = 'ref1'/>
   </uml2.Kernel.Type._typedElementOfType>
   <uml2.Kernel.Enumeration.ownedLiteral>
      <uml2.Kernel.EnumerationLiteral xmi.id = 'el1'
      name = 'male'/>
      <uml2.Kernel.EnumerationLiteral xmi.id = 'el2'
      name = 'female'/>
   </uml2.Kernel.Enumeration.ownedLiteral>
</uml2.Kernel.Enumeration>

<uml:EnumLiteralExp xmi.id = 'ele1' name = 'Gender'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.Enumeration xmi.idref = 'en1'/>
   </uml2.Kernel.TypedElement.type>
   <uml:EnumLiteralExp.referredEnumLiteral>
      <uml2.Kernel.EnumerationLiteral xmi.idref = 'el1'/>
   </uml:EnumLiteralExp.referredEnumLiteral>
</uml:EnumLiteralExp>
```

Definition 15.10: XMI representation for an enumeration literal

15.4.3.5 Collection literals

Collection literals provide the possibility of use collections as values to use inside OCL expressions. To show how to convert a collection literal to the OCL metamodel we will use an example of a Bag literal: `Bag{1, 2 .. 5, 6}`.

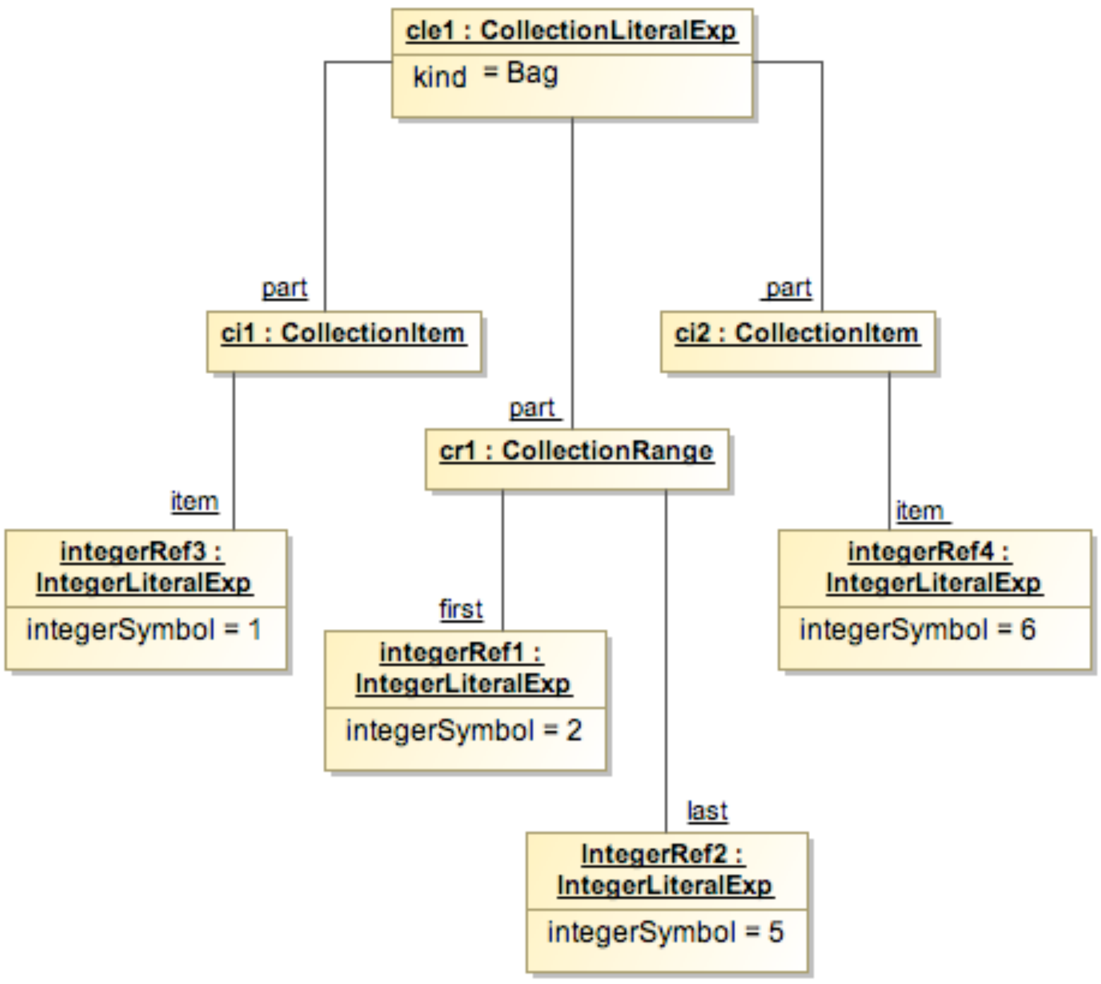


Figure 15.12: Instantiation for a Bag literal expression

Such Bag literal has three members in its definition. First and last ones are CollectionItem instances whereas the second one is a CollectionRange. We can see at previous image the instantiation inside the OCL metamodel for such Bag. It is important to note that the kind of the collection is stored inside the correspondent attribute of the CollectionLiteralExp. Allowed values for it are Bag, Set, OrderedSet and Sequence.

```
<uml:CollectionLiteralExp xmi.id = 'cle1' kind = 'Bag'>
   <uml2.Kernel.TypedElement.type>
      <uml:BagType xmi.idref = 'bagTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CollectionLiteralExp.part>
```

```
      <uml:CollectionItem xmi.idref = 'ci1'/>
      <uml:CollectionRange xmi.idref = 'cr1'/>
      <uml:CollectionItem xmi.idref = 'ci2'/>
   </uml:CollectionLiteralExp.part>
</uml:CollectionLiteralExp>

<uml:CollectionRange xmi.id = 'cr1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CollectionLiteralPart.collectionLiteralExp>
      <uml:CollectionLiteralExp xmi.idref = 'cle1'/>
   </uml:CollectionLiteralPart.collectionLiteralExp>
   <uml:CollectionRange.first>
      <uml:IntegerLiteralExp xmi.idref = 'integerRef1'/>
   </uml:CollectionRange.first>
   <uml:CollectionRange.last>
      <uml:IntegerLiteralExp xmi.idref = 'integerRef2'/>
   </uml:CollectionRange.last>
</uml:CollectionRange>

<uml:CollectionItem xmi.id = 'ci1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CollectionLiteralPart.collectionLiteralExp>
      <uml:CollectionLiteralExp xmi.idref = 'cle1'/>
   </uml:CollectionLiteralPart.collectionLiteralExp>
   <uml:CollectionItem.item>
      <uml:IntegerLiteralExp xmi.idref = 'integerRef3'/>
   </uml:CollectionItem.item>
</uml:CollectionItem>

<uml:CollectionItem xmi.id = 'ci2'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CollectionLiteralPart.collectionLiteralExp>
      <uml:CollectionLiteralExp xmi.idref = 'cle1'/>
   </uml:CollectionLiteralPart.collectionLiteralExp>
   <uml:CollectionItem.item>
      <uml:IntegerLiteralExp xmi.idref = 'integerRef4'/>
   </uml:CollectionItem.item>
</uml:CollectionItem>
```

Definition 15.11: XMI representation for a Bag collection literal

At previous XMI we can observe the direct representation of the OCL metamodel instances showed before for our Bag literal example.

15.4.3.6 Tuple literals

A tuple literal expression provides the possibility of use a multi-attribute structure inside the OCL expressions. In the next example of instantiation we will use a tuple literal expression like this: `Tuple{a:Integer=9, b:String='James'}`

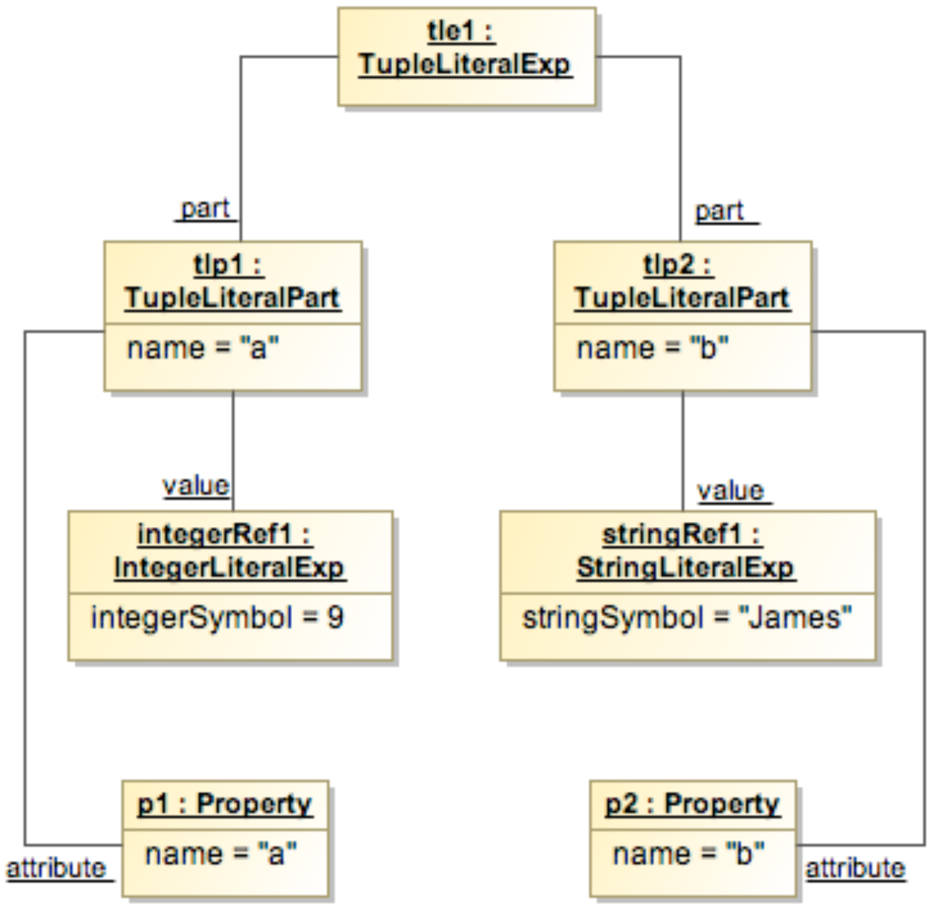


Figure 15.13: Instantiation of a tuple literal expression

As explained when we described the OCL metamodel, the elements of a TupleLiteralExp are instances of the TupleLiteralPart metaclass. Furthermore, such instances store the value of the element in an OclExpression and its definition in an instance of Property. Following these rules, in the previous image we show the instantiation for the tuple literal expression presented as example.

```
<uml:TupleLiteralExp xmi.id = 'tle1'>
   <uml2.Kernel.TypedElement.type>
      <uml:TupleType xmi.idref = 'tupleTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:TupleLiteralExp.part>
      <uml:TupleLiteralPart xmi.idref = 'tlp1'/>
      <uml:TupleLiteralPart xmi.idref = 'tlp2'/>
   </uml:TupleLiteralExp.part>
</uml:TupleLiteralExp>

<uml:TupleLiteralPart xmi.id = 'tlp2' name = 'b'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'stringTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:TupleLiteralPart.tupleLiteralExp>
      <uml:TupleLiteralExp xmi.idref = 'tle1'/>
   </uml:TupleLiteralPart.tupleLiteralExp>
   <uml:TupleLiteralPart.value>
      <uml:StringLiteralExp xmi.idref = 'stringRef1'/>
   </uml:TupleLiteralPart.value>
   <uml:TupleLiteralPart.attribute>
      <uml2.Kernel.Property xmi.idref = 'p2'/>
   </uml:TupleLiteralPart.attribute>
</uml:TupleLiteralPart>

<uml:TupleLiteralPart xmi.id = 'tlp1' name = 'a'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:TupleLiteralPart.tupleLiteralExp>
      <uml:TupleLiteralExp xmi.idref = 'tle1'/>
   </uml:TupleLiteralPart.tupleLiteralExp>
   <uml:TupleLiteralPart.value>
      <uml:IntegerLiteralExp xmi.idref = 'integerRef1'/>
   </uml:TupleLiteralPart.value>
   <uml:TupleLiteralPart.attribute>
      <uml2.Kernel.Property xmi.idref = 'p1'/>
   </uml:TupleLiteralPart.attribute>
</uml:TupleLiteralPart>
```

Definition 15.12: XMI representation for a tuple literal

The XMI conversion is very similar than the instantiation of the diagram. It is important to understant how the elements are linked here using identifiers as reference.

15.4.4 LET EXPRESSIONS

Let expressions allow us to define variables that will be used inside OCL expressions in order to improve the readability of them. At next image we show the instantiation for a let expression with only one variable defined. Remember that it is possible to specify more than one variable in a let expression. For this case the instantiation changes a little due to each variable is placed in a LetExp and the set of LetExps are chained through the *in* relation. Therefore if we have two variables declared in the same let expression, first one is placed in a LetExp whose in expression is another LetExp that owns the second variable.

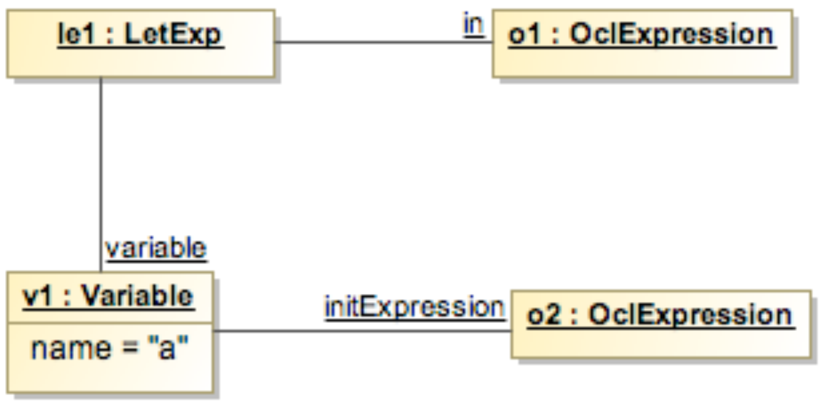


Figure 15.14: Instantiation for a let expression

It is important to note that at this image we have used OclExpression like instances but it is a intentioned error. In real let expressions the *in* expression and the *initExpression* of a variable are both concrete expressions. For example, in the XMI representation is possible to see that the *in* expression is an OperationCallExp and the *initExpression* of the declared variable is an IntegerLiteralExp.

```
<uml:LetExp xmi.id = 'le1' name = 'Let a'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'inTypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:LetExp.in>
      <uml:OperationCallExp xmi.idref = 'inRef'/>
   </uml:LetExp.in>
   <uml:LetExp.variable>
      <uml:Variable xmi.idref = 'v1'/>
   </uml:LetExp.variable>
</uml:LetExp>

<uml:Variable xmi.id = 'v1' name = 'a'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'integerRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:Variable.initExpression>
      <uml:IntegerLiteralExp xmi.idref = 'ileRef'/>
   </uml:Variable.initExpression>
   <uml:Variable.letExp>
      <uml:LetExp xmi.idref = 'le1'/>
   </uml:Variable.letExp>
</uml:Variable>
```

Definition 15.13: XMI representation for a let expression

15.4.5 IF EXPRESSIONS

The if expressions are useful constructions that provides a simple instructions flow control. It is important to remember here the notation for these expressions, so in the next example we can observe it.

If expression:

```
if <boolean condition>
  then <expression>
  else <expression>
endif
```

Example 15.12: Notation for if expressions

Therefore the instantiation of an if expression consists in an instance of IfExp metaclass linked with the three OclExpressions that represents the condition, the then branch and the else branch.

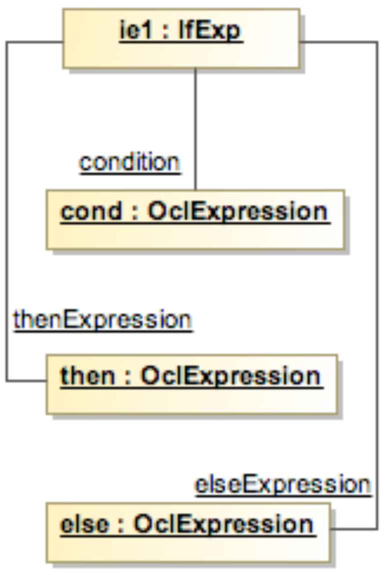


Figure 15.15: Instantiation for an if expression

Finally we can observe in the XMI representation below that each of these three OclExpression instances is an instance of a metaclass that is subclass of OclExpression.

```
<uml:IfExp xmi.id = 'ie1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'datatypeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:IfExp.condition>
      <uml:BooleanLiteralExp xmi.idref = 'condRef'/>
   </uml:IfExp.condition>
   <uml:IfExp.thenExpression>
      <uml:BooleanLiteralExp xmi.idref = 'thenRef'/>
   </uml:IfExp.thenExpression>
   <uml:IfExp.elseExpression>
      <uml:BooleanLiteralExp xmi.idref = 'elseRef'/>
   </uml:IfExp.elseExpression>
</uml:IfExp>
```

Definition 15.14: XMI representation for an if expression

15.4.6 ITERATOR EXPRESSIONS

Iterators, like forAll, select, exists, one, are constructions that provide the functionality of process all the elements of a source collection. Remember that all of them must be used preceded by the arrow -> operator as was explained earlier.

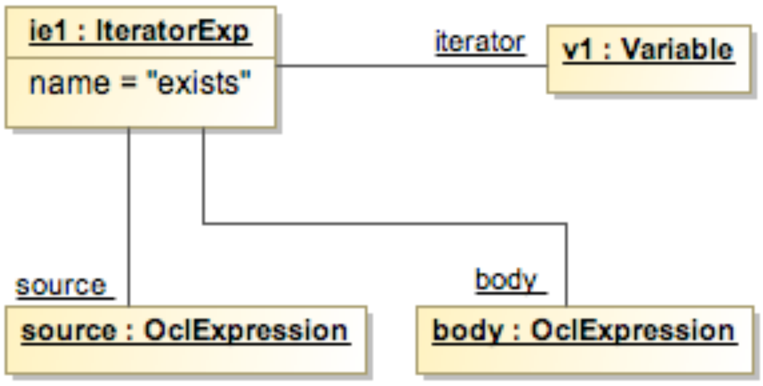


Figure 15.16: Instantiation for an iterator expression

For an iterator expression we can use more than one iterator variable. In such case all iterator variables would appear in both the previous image and the next XMI representation.

```
<uml:IteratorExp xmi.id = 'ie1' name = 'exists'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'booleanRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CallExp.source>
      <uml:OperationCallExp xmi.idref = 'allInstancesRef'/>
   </uml:CallExp.source>
   <uml:LoopExp.body>
      <uml:OperationCallExp xmi.idref = 'oceRef'/>
   </uml:LoopExp.body>
   <uml:LoopExp.iterator>
      <uml:Variable xmi.idref = 'v1'/>
   </uml:LoopExp.iterator>
</uml:IteratorExp>
```

Definition 15.15: XMI representation for an iterator expression

15.4.7 ITERATE EXPRESSIONS

Iterate expressions are a general construction similar than iterator expressions. They can be used in the same places than iterators and also can have more than one iterator variable.

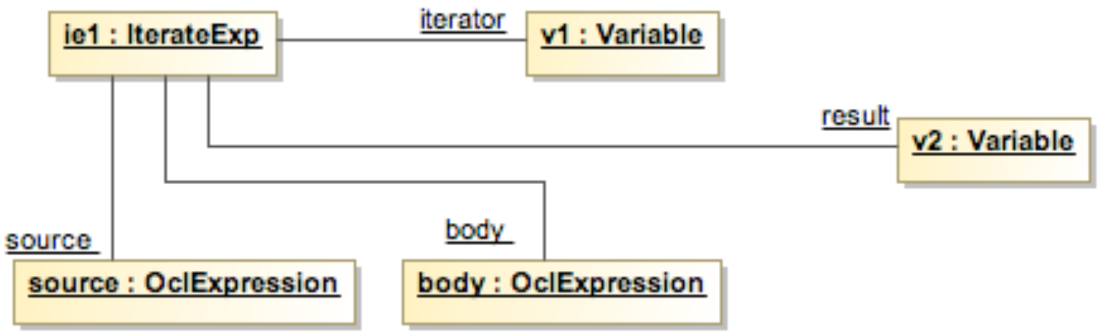


Figure 15.17: Instantiation for an iterate expression

The main difference between iterator instantiation is that iterate constructions have a result variable that specifies the result of the expression. Therefore, we must store it.

```
<uml:IterateExp xmi.id = 'iexp1' name = 'iterate'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'typeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CallExp.source>
      <uml:PropertyCallExp xmi.idref = 'pceRef'/>
   </uml:CallExp.source>
   <uml:LoopExp.body>
      <uml:OperationCallExp xmi.idref = 'oceRef'/>
   </uml:LoopExp.body>
   <uml:LoopExp.iterator>
      <uml:Variable xmi.idref = 'v1'/>
   </uml:LoopExp.iterator>
   <uml:IterateExp.result>
      <uml:Variable xmi.idref = 'v2'/>
   </uml:IterateExp.result>
</uml:IterateExp>
```

Definition 15.16: XMI representation for an iterate expression

15.4.8 PROPERTY CALL EXPRESSIONS

Property call expressions allow us to access to the attributes or association ends from a source class. To use them we must follow the dot notation as in the next example.

Access an attribute of class:

```
source.attributeName
```

Access an association end from a class:

```
source.associationEndName
source.classNameInLowerCase
```

a class name in lower case can be used to access association ends without rolename

Example 15.13: Notation for property call expressions

To instantiate a property call expression we have to create a PropertyCallExp instance and then link it to the property instance to access and to the previous source expression like in the next image.

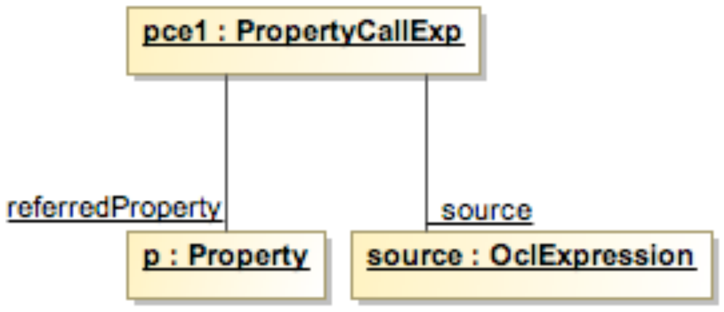


Figure 15.18: Instantiation for a property call expression

Once we have done this instantiation, the representation of it into the XMI language owned by the Eina GMC environment is shown in the next piece of code.

```
<uml:PropertyCallExp xmi.id = 'pce1'>
   <uml2.Kernel.TypedElement.type>
      <uml2.Kernel.DataType xmi.idref = 'typeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CallExp.source>
      <uml:VariableExp xmi.idref = 've1'/>
   </uml:CallExp.source>
   <uml:PropertyCallExp.referredProperty>
      <uml2.Kernel.Property xmi.idref = 'p1'/>
   </uml:PropertyCallExp.referredProperty>
</uml:PropertyCallExp>
```

Definition 15.17: XMI representation for a property call expression

15.4.9 Association Class Call Expressions

Similarly than property call expressions, association class call expressions allow us to access an association class from one of the members of such association. It is necessary to use the same dot notation as before.

Access an association class from a member of such association:

```
source.associationClassNameInLowerCase
```

Example 15.14: Notation for association class call expressions

In this case, we have to create an instance of AssociationClassCallExp to represent the association class call expression processed by our processor.

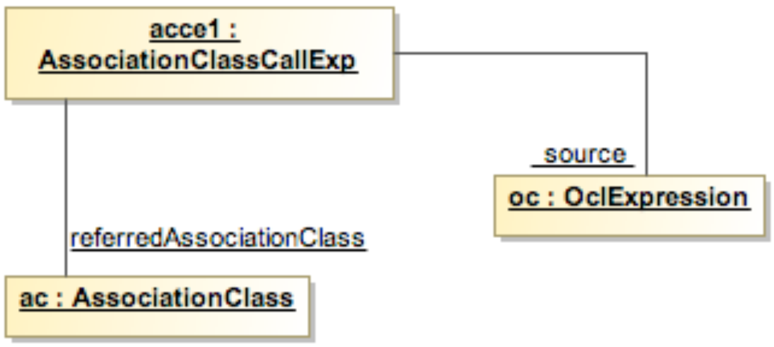


Figure 15.19: Instantiation for an association class call expression

As we can observe in the previous image, such AssociationClassCallExp instance is linked with an AssociationClass and also with a source expression. Therefore the representation into XMI language is a direct conversion.

```
<uml:AssociationClassCallExp xmi.id = 'acce1'>
   <uml2.Kernel.TypedElement.type>
      <uml:BagType xmi.idref = 'typeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CallExp.source>
      <uml:PropertyCallExp xmi.idref = 'pce2'/>
   </uml:CallExp.source>
   <uml:AssociationClassCallExp.referredAssociationClass>
      <uml2.AssociationClasses.AssociationClass xmi.idref = 'ac'/>
   </uml:AssociationClassCallExp.referredAssociationClass>
</uml:AssociationClassCallExp>
```

Definition 15.18: XMI representation for an association class call expression

15.4.10 Operation call expressions

In the OCL 2.0 language it is important to call operations. In the next example we can see the notation to do this task.

Use a mathematical or logical binary operation:

```
13 + 25
```

Use a mathematical or logical unary operation:

```
not true
```

Calling an operation of class with return:

```
source.operationName(list of parameters)
```

Calling an OCL specific dot operation:

```
source.operationName(list of parameters)
p.oclAsType(Customer)
```

Calling an OCL specific arrow operation:

```
source->operationName(list of parameters)
Set{1,2,3,4}->size()
```

Example 15.15: Notation for operation call expressions

In all cases an operation call expressions have a source expression where it is applied and an operation. If the operation call has parameters, they are stored into the *argument* field of OperationCallExp instance.

Furthermore if we are calling an operation with a unary operator, the operand will be stored as the source of the operator. In addition, when a binary operation is called (like in 3 + 2), the left operand is stored as *source* of the OperationCallExp instance whereas the right operand is stored like a parameter in the *argument* field.

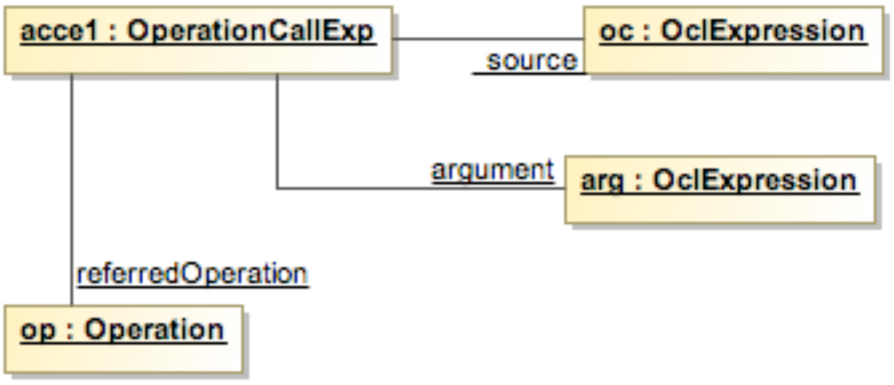


Figure 15.20: Instantiation for an operation call expression

In the previous image we can observe the instantiation of an operation call expression following the rules explained before, and then the conversion to the XMI representation for this OperationCallExp instance is shown at next piece of code.

```
<uml:OperationCallExp xmi.id = 'oce1'>
   <uml2.Kernel.TypedElement.type>
      <uml:SetType xmi.idref = 'typeRef'/>
   </uml2.Kernel.TypedElement.type>
   <uml:CallExp.source>
      <uml:VariableExp xmi.idref = 've1'/>
   </uml:CallExp.source>
   <uml:OperationCallExp.argument>
      <uml:IntegerLiteralExp xmi.idref = 'ileRef'/>
   </uml:OperationCallExp.argument>
   <uml:OperationCallExp.referredOperation>
      <uml2.Kernel.Operation xmi.idref = 'op1'/>
   </uml:OperationCallExp.referredOperation>
</uml:OperationCallExp>

<uml2.Kernel.Operation xmi.id = 'op1' name = 'including'
isLeaf = 'false' isStatic = 'false' upper = '0' isUnique = 'false'
 isQuery = 'false' lower = '0' isOrdered = 'false'>
   <uml2.Kernel.Operation.referringExp>
      <uml:OperationCallExp xmi.idref = 'oce1'/>
   </uml2.Kernel.Operation.referringExp>
</uml2.Kernel.Operation>
```

Definition 15.19: XMI representation for an operation call expression

15.5 INVERSE CONVERSION

At previous section we have studied the conversion process of our processor in order to transform textual OCL 2.0 constraints into instances of the OCL 2.0 metamodel. Now, we will introduce the inverse conversion that returns such metamodel instances into their original textual constraints.

To do this task we decided to implement another visitor class extending ReflectiveVisitor explained before. With this architecture we follow a divide and conquer strategy that simplifies the processing of the metamodel instances until become textual constraints.

A new package is introduced here inside the Eina GMC project called *converterOCL*, which contains the ConverterOCL.java file representing the inverse converter component.

To know how it works we will adapt the Example 15.1 related to the Type Check component with the own methods of this new component. We can see it in the next example.

There exists a general method to be called for every element to process:

```
public String convert (Element e)
```

Such method can be called for an IfExp element:

```
// Imagine that conv is an instance of ConverterOCL class
// and ifexp an instance of IfExp metaclass
conv.convert(ifexp);
```

This invocation implies that Reflective Visitor searches and then has to find a method with the next signature:

```
public String conversion(IfExp exp)
```

There exists a conversion method for every element of the metamodel. This method is called for the ifexp instance of IfExp metaclass. Inside this method we could do as follows:

```
OclExpression cond = exp.getCondition();
OclExpression thenBranch = exp.getThenExpression();
OclExpression elseBranch = exp.getElseExpression();
String condTxT = convert(cond);
String thenTxT = convert(thenBranch);
String elseTxT = convert(elseBranch);
```

Using this kind of visitor we only have to obtain the attributes of IfExp metaclass and then call the convert method of each one to obtain their textual representation. This recursive strategy simplifies the inverse conversion process.

On this example, last step consists in to return the textual representation of the whole IfExp.

```
return “if “ + condTxT + “ then “ + thenTxT +
                         ” else “ + elseTxT + ” endif”;
```

Example 15.16: Usage of the TypeCheck class as a subclass of the ReflectiveVisitor

Therefore to change the behaviour of a concrete conversion we only have to change the conversion method of the element in question. This strategy provides an easier changeability to our processor.

15.6 DELETE CONSTRAINTS

Inside package facadeOCL we can find the DeleteVisitor.java file that contains a new ReflectiveVisitor implementation. This component provides the possibility of delete a constraint previously processed and now instantiated into the OCL 2.0 metamodel of the metadata repository owned by the Eina GMC conceptual modeling environment.

As we have seen when both TypeCheck and ReflectiveVisitor were explained, this kind of visitor uses the Java Reflective API in order to discover the class type of the elements to process.

Therefore, to explain the behaviour of the DeleteVisitor we will adapt the Example 15.1 with the own methods of this new component.

There exists a general method to be called for every element to process:

```
public boolean remove (Element e)
```

Such method can be called for an IfExp element:

```
// imagine that dv is an instance of DeleteVisitor class
// and ifexp an instance of IfExp metaclass
dv.remove(ifexp);
```

This invocation implies that Reflective Visitor searches and then has to find a method with the next signature:

```
public boolean delete(IfExp exp)
```

This method is called for the ifexp instance of IfExp metaclass. Inside this method we could do as follows:

```
OclExpression cond = exp.getCondition();
OclExpression thenBranch = exp.getThenExpression();
OclExpression elseBranch = exp.getElseExpression();
boolean condDeleted = remove(cond);
boolean thenDeleted = remove(thenBranch);
boolean elseDeleted = remove(elseBranch);
if (condDeleted && thenDeleted && elseDeleted) {
   exp.refDelete();
   return true
}
```

Using this kind of visitor we only have to obtain the attributes of IfExp metaclass and then call the remove method for each one to delete them. This recursive strategy simplifies the erasure process.

On this example, last step consists on to delete the IfExp instance from the metadata repository. To do this we use the refDelete operation owned by each element of the Eina GMC's implementation for the UML and OCL metamodels.

Example 15.17: Usage of the DeleteVisitor class as a subclass of the ReflectiveVisitor

Finally, there exists another way of delete a constraint. Similarly than we did with the usage of the processor to parse constraints, we have provided an additional facade method to simplify the usage of the DeleteVisitor. Such method is placed inside the ParserFacade.java file of the facadeOCL package.

To delete a constraint using this method it is mandatory to know the name of the constraint. It is important to remember here that every kind of constraint has the possibility to be named using a name inside the structure of its context. At next example we will show some examples of named constraints and then the method to delete them.

A list of constraints previously parsed

```
context Employee
 inv c1 : salary > 1000
 inv c2 : age > 0
context Employee::hasDescendants:Boolean
 derive c3 : self.descendants->size() > 0
context Department::getEmployeeNames():Set(String)
 post c4: result = self.employee.name->asSet()
```

```
context Person::address:String
 init: ‘no address’
```

Then, to delete these constraints we must execute:

```
// Suppose that p is the actual Project instance with the
// XMI model containing the previous constraints
ParserFacade pf = new ParserFacade();
pf.deleteOclExpression(“c1”,p);
pf.deleteOclExpression(“c2”,p);
pf.deleteOclExpression(“c3”,p);
pf.deleteOclExpression(“c4”,p);
```

It is important to note that last constraint is not possible to delete with this method because it does not contains a name to be identified.

Example 15.18: Usage of DeleteVisitor with the deleteOclExpression method owned by the ParserFacade class

We recommend using a name in order to identify each constraint and then use the facade method instead of directly the method of DeleteVisitor class. It is also important to note that we are only able to delete complete constraints and not only single elements or expressions. It is due to avoid inconsistent constraints.

15.7 XMICONVERTER AND OUR OCL PROCESSOR

When we introduced the Eina GMC conceptual modeling environment and its complmentary tools we showed the XMIConverter as a tool that converts XMI files from a format to another one. In our case we describe how to use it to convert a Poseidon for UML XMI file into a file to be used by Eina GMC.

To provide a graphical way for modelling we said that we could construct a conceptual schema directly in Poseidon for UML and then convert it to the Eina GMC format. In order to help users to work with our OCL processor we have extended the XMIConverter with a new feature that may be explained.

Poseidon for UML provides a special section inside the specification menu of each UML class to write constraints over such class. But we checked that such information is not stored inside the XMI file when we export our diagrams as explained in section 5.5.1 of the chapter 5. Furthermore, Poseidon does not verify wether such constraints are well-formed expressions.

Therefore, we have have included a new conversion inside the XMIConverter that deals with comment elements of Poseidon diagrams. Now, if we want to write an OCL 2.0 constraint directly in a diagram of Poseidon we only have to write it inside a comment of such diagram. XMIConverter searches for those comments and gets their contents in order to pass them to our processor of OCL 2.0 expressions.

So we have made a link between the Poseidon, the XMIConverter and our tool to parse OCL constraints directly inside the conversion process of the XMIConverter from Poseidon format to EinaGMC format.

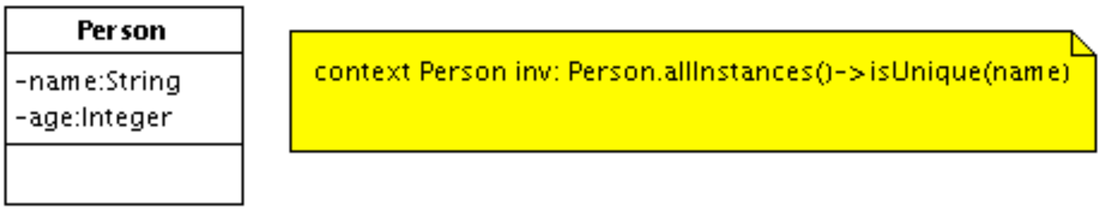


Figure 15.21: Usage of constraints inside Poseidon to be checked by our OCL processor

In the previous figure we can see how to use a comment with a constraint. It is important to note that if we decide to write our constraints directly in the Poseidon diagram, errors found during the checking process made by our tool will be thrown during the conversion process of the XMIConverter. Therefore, we will have to solve such errors directly inside the comments.

CASE STUDY: DBLP 16

16 CASE STUDY: DBLP

16.1 INTRODUCTION

The aim of this chapter is to show the instantiation result of the conversion process made by our processor for OCL 2.0 expressions. We will select a little set of constraints written in a well-known case study made inside the GMC Research Group [GMCw].

Our tool will process such constraints and then we will show a diagram with the resultant instances of this process. If any problem is found during the conversion it will be explained here in order to help future users of the processor how to work with it.

Finally, it is important to emphasize that this chapter will not be a correctness checker for all constraints written in the case study that we will present here.

16.2 THE DBLP SYSTEM

The case study we will use in this chapter is the DBLP Case Study written by Elena Planas and Antoni Olivé [PO06]. It contains parts of the conceptual schema of the DBLP system, written in UML.

DBLP [DBLPw], also known as Digital Bibliography & Library Project, is a computer science website hosted at Universität Trier, in Germany. It was originally a database and logic programming bibliography site, and has existed at least since the 1980s. DBLP listed more than one million articles on computer science in March 2008.

First of all we will show the conceptual schema presented in the case study introduced before. Such schema deals with persons and their publications, which may be edited books or authored publications such as authored books, book chapters and journal papers.

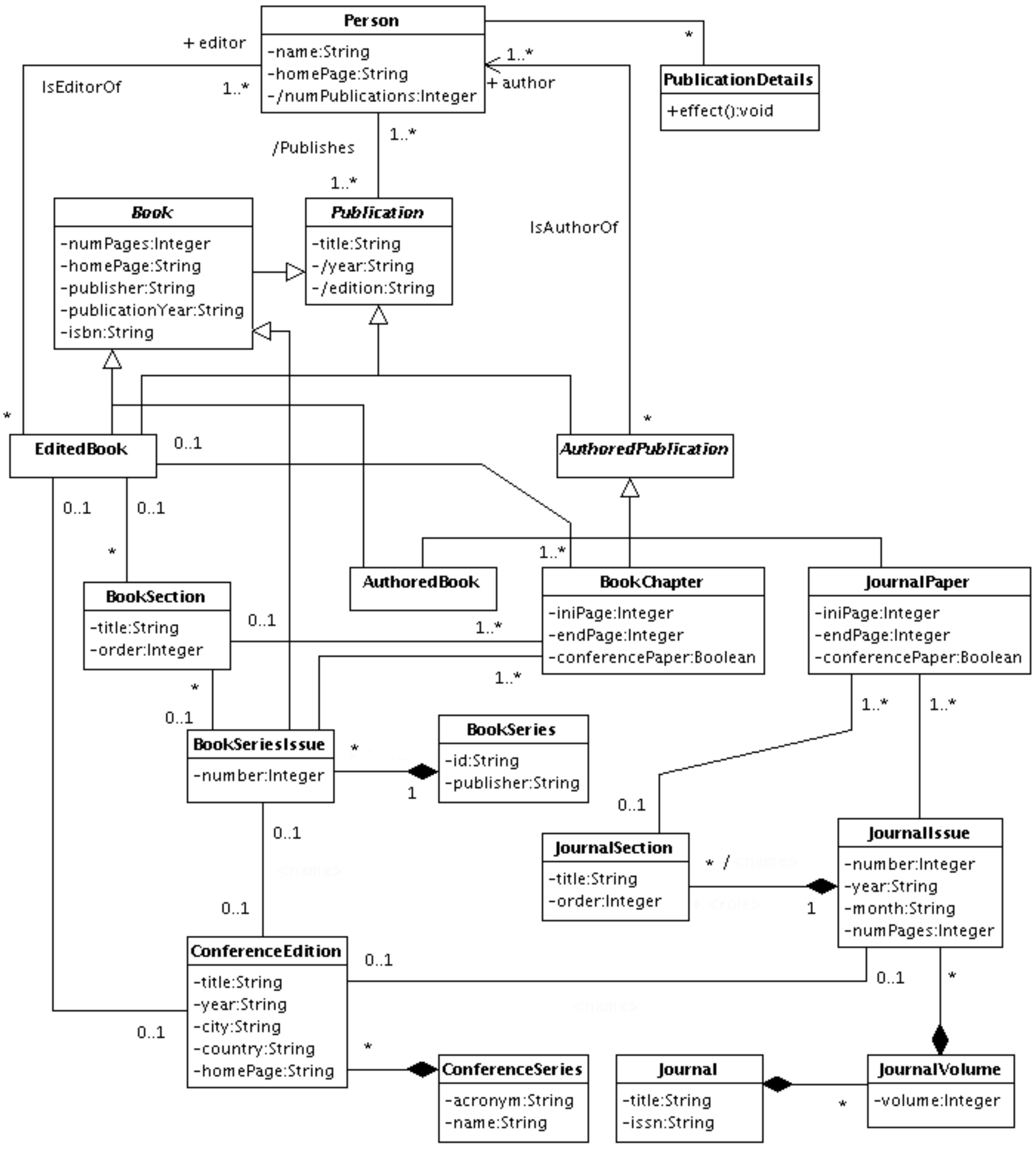


Figure 16.1: DBLP model used in the case study

To be honest we must affirm that this model has some changes in respect with the original used in the case study. We have changed the Year type of year attributes for the String data type because our processor does not support user-defined data types by now.

Furthermore, we have changed aggregate associations to normal associations because the converter tool that converts XMI models from Poseidon format to Eina GMC format does not support this kind of associations.

After that, we export this conceptual model written in Poseidon for UML into the XMI format. We have followed the steps shown at chapter 5, when we explained the Eina GMC environment.

Once we have the model converted into the Eina GMC format we can start the process of conversion for our selected constraints. Such constraints are written inside the document of the case study.

16.3 CONVERSION EXAMPLES

In this section we will introduce a minimal set of constraints of the DBLP case study document [PO06] with their instantiation diagrams of the instances created by our processor of OCL 2.0 expressions.

16.3.1 IDENTIFICATION CONSTRAINTS

First constraint to process is a simple identification constraint that indicates wich attribute is the primary key, using database language, of the contextual class.

```
context Person inv nameIsKey: -- First constraint
  Person.allInstances()->isUnique(name)
```

Example 16.1: First constraint to check

In this case we select this constraint that indicates each Person of the system must have a different name.

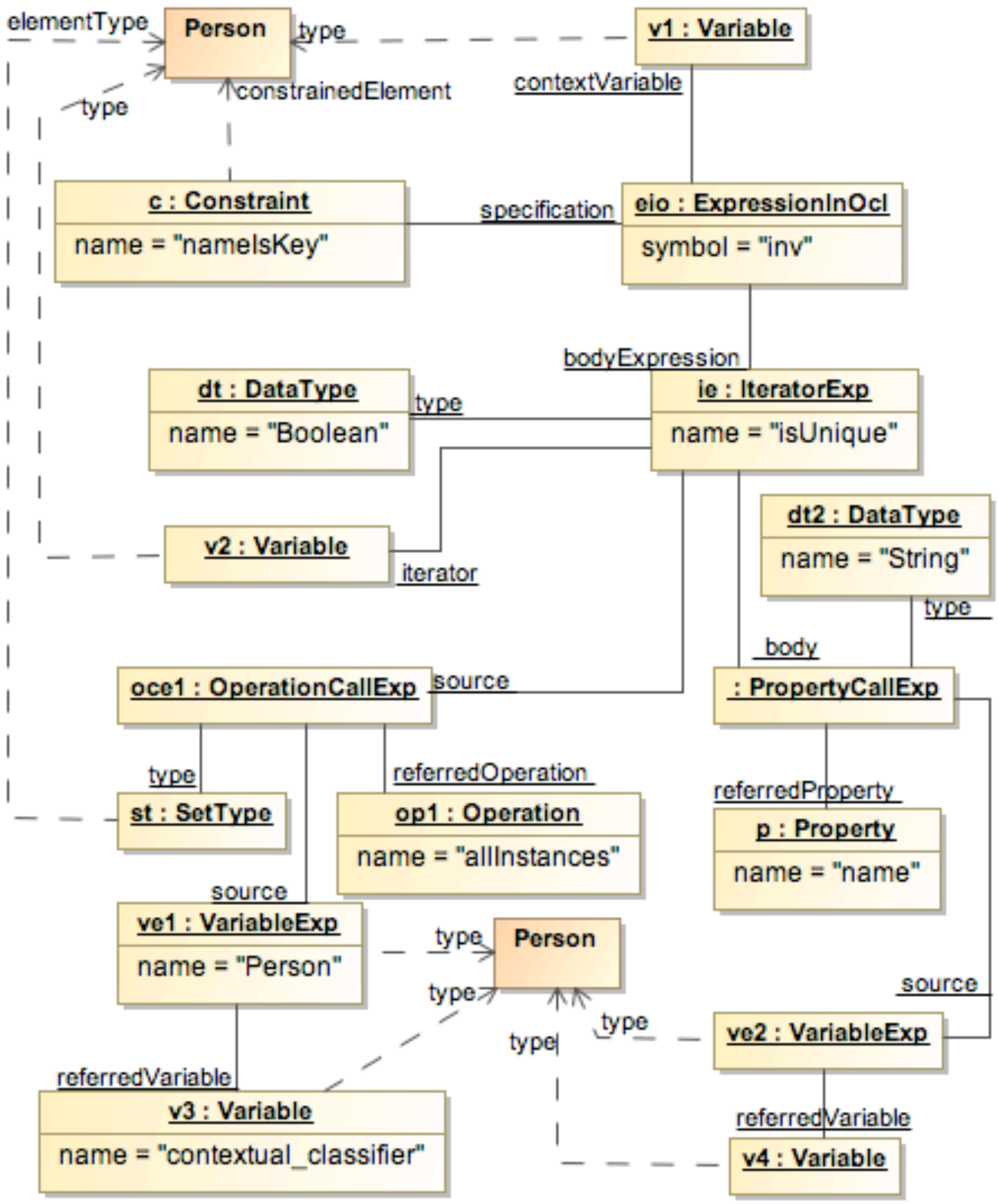


Figure 16.2: Complete instantiation for constraint "nameIsKey"

The output instantiation that results of the parsing process made by our processor of OCL expressions is shown at this image.

The main expression of the constraint is the *isUnique* iterator expression that has the application of the *allInstances* operation to the Person class as source. Furthermore, the body of this iterator is a property call expression that references the *name* Property of such class.

As we can see, even a simple constraint hides a complexity process of instantiation that must be done properly.

16.3.2 INTEGRITY CONSTRAINTS

The second constraint selected is a common integrity constraint that allows modellers to set an invariant to their models in order to specify a situation that always must be true.

```
-- Second constraint:
-- The pages of the papers (JournalPaper) published in a journal
-- issue (JournalIssue) do not overlap among them.
context JournalIssue inv correctPagination:
  self.journalPaper->forAll(p1,p2|p1<>p2 implies
       p1.iniPage > p2.endPage or p2.iniPage > p1.endPage)
```

Example 16.2: Second constraint to check

Once we have processed such constraint we do not find any error because such constraint is very common. It contains a main *forAll* iterator that controls pages of papers.

We can use this structure of forAll iterator with two iterator variables that are compared in order to verify a constraint over attributes of these variables. In this case we control that pages of papers do not overlap among them.

To show the instantiation of this constraint we will use a hierarchical structure instead of an instance diagram in order to improve the understandability of the output XMI thrown by our processor. We have implemented a new Java class called

HumanOutputGenerator that generates our desired hierarchical structure of constraints placed in a XMI file. The output for the instances created for this second constraint is shown in the next tree.

```
Constraint correctPagination
  ConstrainedElement
    Class JournalIssue
  Specification
    ExpressionInOcl inv
      ContextVariable
        Variable
          Type
            Class JournalIssue
      BodyExpression
        IteratorExp forAll
          Iterator
            Variable p1
              Type
                Class JournalPaper
            Variable p2
              Type
                Class JournalPaper
          Source
            PropertyCallExp
              ReferredProperty
                Property
                  Type
                    Class JournalPaper
              Source
                VariableExp
                  ReferredVariable
                    Variable self
                      Type
                        Class JournalIssue
                  Type
                    Class JournalIssue
              Type
                Class JournalPaper
          Body
            OperationCallExp
              ReferredOperation
                Operation or
              Argument
                OperationCallExp
                  ReferredOperation
                    Operation >
                  Argument
                    PropertyCallExp
                      ReferredProperty
                        Property endPage
                          Type
                            DataType Integer
                      Source
                        VariableExp
                          ReferredVariable
                            Variable p1
                              Type
                                Class JournalPaper
                          Type
                            Class JournalPaper
                      Type
                        DataType Integer
                  Source
                    PropertyCallExp
                      ReferredProperty
                        Property iniPage
                          Type
                            DataType Integer
                      Source
                        VariableExp
                          ReferredVariable
                            Variable p2
                              Type
                                Class JournalPaper
                          Type
                            Class JournalPaper
                      Type
                        DataType Integer
                  Type
                    DataType Boolean
              Source
                OperationCallExp
                  ReferredOperation
                    Operation implies
                  Argument
                    OperationCallExp
                      ReferredOperation
                        Operation >
                      Argument
                        PropertyCallExp
                          ReferredProperty
                            Property endPage
                              Type
                                DataType Integer
                          Source
                            VariableExp
                              ReferredVariable
                                Variable p2
                                  Type
                                    Class JournalPaper
                              Type
                                Class JournalPaper
                          Type
                            DataType Integer
                      Source
                        PropertyCallExp
                          ReferredProperty
                            Property iniPage
                              Type
                                DataType Integer
                          Source
                            VariableExp
                              ReferredVariable
                                Variable p1
                                  Type
                                    Class JournalPaper
                              Type
                                Class JournalPaper
                          Type
                            DataType Integer
```

```
                        Type
                          DataType Boolean
                    Source
                      OperationCallExp
                        ReferredOperation
                          Operation <>
                        Argument
                          VariableExp
                            ReferredVariable
                              Variable p2
                                Type
                                  Class JournalPaper
                            Type
                              Class JournalPaper
                        Source
                          VariableExp
                            ReferredVariable
                              Variable p1
                                Type
                                  Class JournalPaper
                            Type
                              Class JournalPaper
                        Type
                          DataType Boolean
                    Type
                      DataType Boolean
                Type
                  DataType Boolean
            Type
              DataType Boolean
```

Example 16.3: Hierachical output for the constraint found in the Example 16.2

We have marked with the same color the same instance and its links in order to improve the understandability of this output.

16.3.3 DERIVATION RULES

Another constraint of the selected set is a derivation rule for a derived attribute of the previous model. In this case this rule uses an if expression to select between two alternatives depending on a condition.

Once we processed this constraint we found a problem with the *publicationYear* attribute of the EditedBook class due to the multiple inheritance of such class.

We use the XMIConverter tool of the Eina GMC environment in order to convert a XMI model in Poseidon format to the Eina GMC format. Such tool has a little problem with models that contains examples of multiple inheritance because only converts one of the superclass links. In our case, our DBLP model contains the EditedBook class as subclass of both Book and Publication classes. But the XMIConverter only stores inside EditedBook class a reference to its superclass Publication but not Book.

```
-- The year of BookChapter publication is the publication year of
-- the book or journal issue that publishes it.
context BookChapter::year:String
  derive: if self.editedBook->notEmpty()
          then self.editedBook.publicationYear
          else self.bookSeriesIssue.publicationYear
          endif
```

Example 16.4: Original third constraint

In the previous constraint we can see the original version of derivation rule and in the constraint below we can observe the change in the attribute due to the XMIConverter problem.

```
-- The year of BookChapter publication is the publication year of
-- the book or journal issue that publishes it.
context BookChapter::year:String
  derive: if self.editedBook->notEmpty()
          then self.editedBook.year --This is the change
          else self.bookSeriesIssue.publicationYear
          endif
```

Example 16.5: Third constraint with changes

We decided to change the constraint in order to be processed because the XMI is not correct due to the inheritance problem of the XMIConverter. In particular, changing the attribute publicationYear for the attribute year, which is inherited from Publication class, implies that the problem with the multiple inheritance is avoided although the constraint is not semantically equal to the original.

Nevertheless, our aim here is to present how the output of the process is made instead of the semantics of the constraints selected. In the next output of the HumanOutputGenerator tool we can observe the instantiation created for the previous constraint.

```
Constraint
  ConstrainedElement
    Property year
      DataType String
  Specification
    ExpressionInOcl derive
      ContextVariable
        Variable
          Type
            Class BookChapter
      BodyExpression
        IfExp
          Condition
            OperationCallExp
              ReferredOperation
                Operation notEmpty
              Source
                PropertyCallExp
                  ReferredProperty
                    Property
                      Type
                        Class EditedBook
                  Source
                    VariableExp
                      ReferredVariable
                        Variable self
                          Type
                            Class BookChapter
                      Type
                        Class BookChapter
                  Type
                    Class EditedBook
              Type
                DataType Boolean
          ThenExpression
            PropertyCallExp
              ReferredProperty
                Property year
                  Type
                    DataType String
              Source
                PropertyCallExp
                  ReferredProperty
                    Property
                      Type
                        Class EditedBook
                  Source
                    VariableExp
                      ReferredVariable
                        Variable self
                          Type
                            Class BookChapter
                      Type
                        Class BookChapter
                  Type
                    Class EditedBook
              Type
                DataType String
          ElseExpression
            PropertyCallExp
              ReferredProperty
                Property publicationYear
                  Type
                    DataType String
              Source
                PropertyCallExp
                  ReferredProperty
                    Property
                      Type
                        Class BookSeriesIssue
                  Source
                    VariableExp
                      ReferredVariable
                        Variable self
                          Type
                            Class BookChapter
                      Type
                        Class BookChapter
                  Type
                    Class BookSeriesIssue
              Type
                DataType String
          Type
            DataType String
```

Example 16.6: Hierarchical output for the constraint found in the Example 16.5

16.3.4 QUERY SPECIFICATION

Finally we introduced in the model of the DBLP system an extra class named PublicationDetails that given a person provides the information of the volume issues and the papers published in each of them.

This information is given through a query method called *effect* that is part of the behavioural schema of the DBLP.

At next description is possible to see the structure of the PublicationDetails class including the *answer* attribute that will contain the information returned by the *effect* method.

```
Class: PublicationDetails
Attributes:
  answer: Set (TupleType
                 (year: Year,
                  yearPublications:
                    Set (TupleType
                            authorsOrEditors: Set(String),
                            title: String,
                            edition: String)))
Operations:
  effect()

context PublicationDetails::effect()
post:
  answer =
   self.person.publication.year -> asSet() -> collect(y |
     Tuple
       {year = y,
        yearPublications =
          self.person.publication -> select (p | p.year = y)
            -> collect (p2 |
                Tuple
                {authorsOrEditors = p2.person.name,
                 title = p2.title,
                 edition = p2.edition})}) -> sortedBy(year)
```

Example 16.7: Specification for PublicationDetails class

Furthermore, we can also observe a postcondition for the *effect* method explaining how to obtain the information required.

Our first step was to define the answer attribute inside the PublicationDetails class because we cannot do it directly through Poseidon for UML.

```
context PublicationDetails
  def answerAttribute:
      answer:Set(
               TupleType(year:String,
                         yearPublications:Set(
                           TupleType(
                                authorsOrEditors:Set(String),
                                title:String,
                                edition:String)))) = Set{}
```

Example 16.8: Fourth constraint to check

In this case we decided to use a definition constraint in which we define such attribute including its type and the value of it as an empty set.

The output generated as instances tree is the hierarchychal structure shown here.

```
Constraint answerAttribute
  ConstrainedElement
    Property answer
      SetType
        ElementType
          TupleType
            OwnedAttribute
              Property year
                Type
                  DataType String
              Property yearPublications
                SetType
                  ElementType
                    TupleType
                      OwnedAttribute
                        Property authorsOrEditors
                          Type
                            SetType
                              ElementType
                                DataType String
                        Property title
                          Type
                            DataType String
                        Property edition
                          Type
                            DataType String
  Specification
    ExpressionInOcl def
      ContextVariable
        Variable
          Type
            Class PublicationDetails
      BodyExpression
        CollectionLiteralExp Set
          Type
            SetType
```

Example 16.9: Hierarchical structure for constraint found in the Example 16.8

Once we processed the postcondition we found some problems related to the collection kinds. It is needed to include to *asSet()* operations after the two collect iterators because the result of them is a Bag instead of a Set collection.

Furthermore, in the original constraint appears a sortedBy iterator to indicate that the resultant Tuples includes in the output Set are ordered by the *year* member of these tuples.

It is an error because as specified in the OCL 2.0 specification document [Obj06], the resultant type of the body of a sortedBy iterator must have defined the "<" comparison operator. In our case, year is a String and it has no such operator defined.

It is important to note here that we previously change the type of year attribute from Year to String because our processor does not support custom DataTypes, therefore it is possible to consider it as a correct usage in the original constraint.

```
context PublicationDetails::effect():Boolean
post pubdetails:
answer = self.person.publication.year->asSet()
  ->collect(y|
          Tuple{year = y,
                yearPublications =
                  self.person.publication->select(p|p.year=y)->
                    collect(p2|
                      Tuple
                      {authorsOrEditors = p2.person.name->asSet(),
                       title=p2.title,
                       edition=p2.edition})->asSet()}
           )->asSet() --we can use ->asSequence()->asSet() instead
```

Example 16.10: Fifth constraint to check

In our case we decided to delete this operation from the constraint or to use a previous *asSequence* operation before the last *asSet* needed for the first *collect* iterator due to the *asSequence* specifies that the result will be ordered although it does not indicates how the sorting is done. The final version of the constraint is shown at previous OCL code.

Finally, it is important to note that the output instance tree for this constraint is too large, so we preferred to do not show it here. The models and selected constraints can be found at my website [Villw] where are free to download and test with the HumanOutputGenerator of our processor of OCL 2.0 expressions.

FUTURE WORK 17

17 FUTURE WORK

17.1 WAYS TO IMPROVE THE PARSER

Once we have exposed how is made our processor of OCL 2.0 expressions and its features, it is the moment of explain some possible extensions or changes to apply this tool in order to improve it.

The Parser Subsystem of the Dresden OCL2 Toolkit has some limitations and recognised bugs. For example, there are problems when navigating from an associative class to their association ends, and we have to remember that such tool only supports UML in its version 1.5.

In our processor we are fixed most of these problems and we have adapted it to the last version of the UML. Nevertheless, our tool is not perfect and has some problems that were introduced in previous examples. Our aim here is to show a general view of these problems and possible solutions for them.

17.1.1 COMPLETE SOME OCL CONSTRUCTIONS

Our tool supports the OCL language in its version 2.0. We have taken care to construct a processor that allows all possible OCL constructions to make constraints, but finally we know that some constructions are not supported.

Principally, we decided to avoid support message expressions, i.e., constructions that indicates the execution of a method in some part of an expression denoted with ^ and ^^ tokens. It was a time problem because our planification was changed due to external problems and we decided to do not support it. It is not a big problem because such constructions are less important than others that are supported by our tool. To support message expressions was not a main objective of our development compared with other parts of the OCL language.

Other constructions not supported are the named-self constructions. It is a low documented construction that allows defining a context construction where the contextual classifier is named impliying that variable *self* is substituted by another variable with the previous name. We can see a little example at next constraint.

```
context p:Person inv: p.age > 0 instead of
context Person inv: self.age > 0
```

Example 17.1: Named self context construction

To solve these problems should not be difficult because we only should to change our grammar, generate the framework through SableCC-Ext and then complete the methods that create the instances of these constructions.

17.1.2 TYPE CONFORMANCE EXTENDED

As explained earlier, the type checking process of our processor has a lack related to the support of user data types. In this first version of our processor we only support the data types specified in the OCL 2.0 specification document, i.e., Integer, Real, String and Boolean data types.

It is possible that users want to create its owned data types, like Date or Hour, but by now we cannot assure that our processor works properly with them. To solve it there are different alternatives. One of them could be to add an extra case in the type checker that would allow work with such types with only basic logical and arithmetic operations like +, -, *, >, <, = or<>.

This implementation has the problem that it is possible to have data types that are not allowed to use with some of these operations, so the processor could have problems with them.

The other solution is to define one operation for each data type. For example, we could have two operations to do multiplications, one for Integers and one for Reals, and if some other user data type need the * operator, they should create a new operation whose parameters would be instances of such types. This solution implies to have repeated operations impliying an increased size of XMI files.

By now, the type checker uses the name of the operation to verify if its operands are correct. The new implementation should check its operands and then verify if the operation name matches with an existing operation that is allowed with the current types of the operands. To support it, we should change a big part of the code of our type checker.

Finally, we have to indicate that the solution of this problem has to be decided by both the users of our processor, and the designers and programmers of it. Once a decision would be taken, we will implement its changes in order to support user data types.

17.1.3 OCL INTERPRETATION

One of the most useful features for a compiler of OCL expressions is to interprete expressions. As we seen when we explained USE [USEw] and MOVA tool [MOVw], it is possible to execute or interprete OCL expressions in order to obtain the result of their application.

To do this, we only have to implement a new visitor class that traverse the instances tree result of the parsing process. Such class should be similar than the classes that we use to delete constraints or to convert them into another format of representation. Therefore, we need a second sweep that works with the instances of a conceptual schema and returns the correct result of the application of OCL constraints previously checked. A possibility could be to study the interpretation process done by USE and to adapt it to our processor.

This feature adds more complexity to the implementation process although it will be well appreciated by users of our tool. Nevertheless, we cannot forget that now this feature is secondary in our planification.

17.1.4 A FULL ENVIRONMENT

We have done improvements in the usability of our processor including simple facade operations that allow us to call our tool directly with a high level of abstraction. Furthermore, we also have adapted the XMIConverter to check constraints placed inside comments of the diagrams made in CASE tools (in our case, Poseidon for UML).

Although our work has been hard in order to improve this discipline, we think that to create a full environment in which users cannot have to write program code to use our tools should be a mandatory future task.

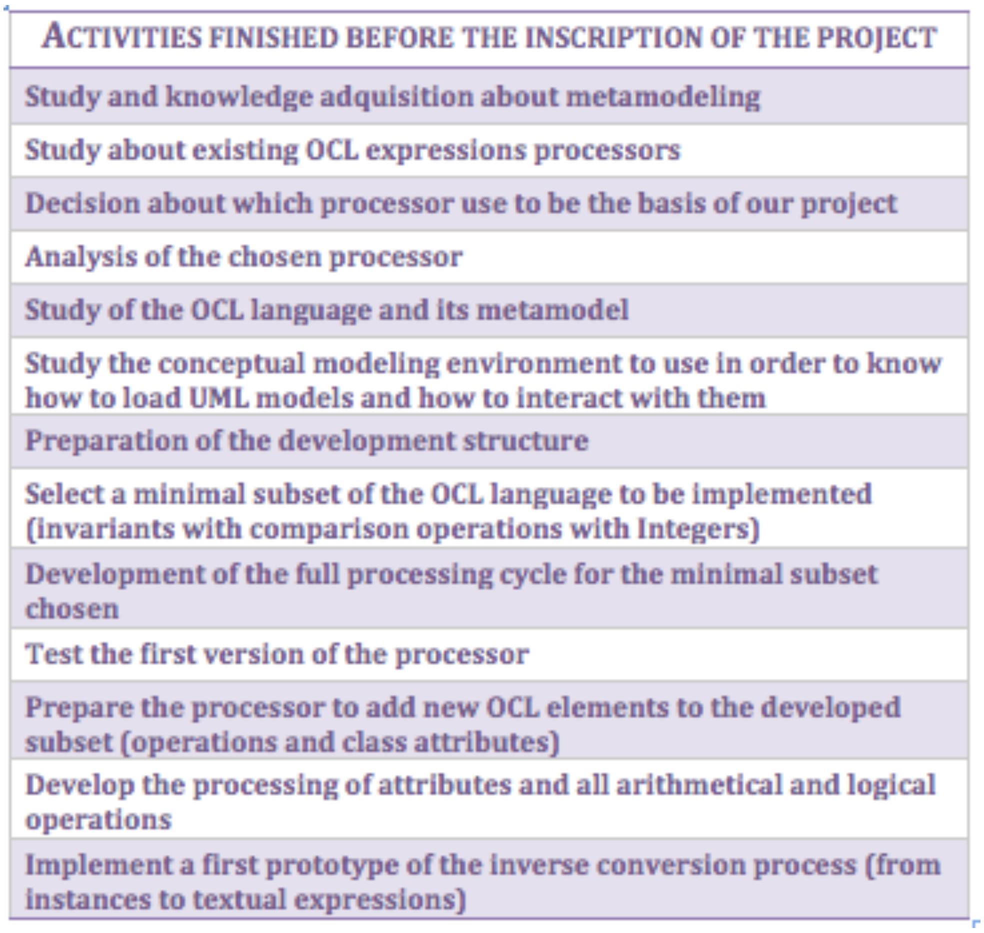
Since Eina GMC tool and its components are growing up, to have a main graphical environment in which construct conceptual schemas, create instances, and write and check OCL constraints would help and animate new users to get in touch and work with a combination of all these tools.

PROJECT PLAN 18

18 PROJECT PLAN

18.1 INTRODUCTION

This project formally began the 31st of January 2008 with its inscription and later registration into the Barcelona School of Informatics. Nevertheless, it is important to note that the study phase was started before it, during the spring quadrimester of 2007, when the first contacts with the director Antoni Olivé occurs and the work plan is established.



ACTIVITIES FINISHED BEFORE THE INSCRIPTION OF THE PROJECT
Study and knowledge adquisition about metamodeling
Study about existing OCL expressions processors
Decision about which processor use to be the basis of our project
Analysis of the chosen processor
Study of the OCL language and its metamodel
Study the conceptual modeling environment to use in order to know how to load UML models and how to interact with them
Preparation of the development structure
Select a minimal subset of the OCL language to be implemented (invariants with comparison operations with Integers)
Development of the full processing cycle for the minimal subset chosen
Test the first version of the processor
Prepare the processor to add new OCL elements to the developed subset (operations and class attributes)
Develop the processing of attributes and all arithmetical and logical operations
Implement a first prototype of the inverse conversion process (from instances to textual expressions)

Table 18.1: Activites finished before the inscription of the project

In the previous table we can see the activities finished before the inscription of the project. As explained in such table, we had implemented a complete version of the final system although it only supported a little subset of OCL elements to construct expressions.

Anyways, although such version had a lot of limitations it helped us in order to verify if our final users agree with the results achieved.

18.2 ESTIMATED PLAN

Here we will show a cronogram with the estimated planification of tasks for the period between the inscription of the project until the presentation of it in the presence of the board of professors that will evaluate it.

In such cronogram we can see both the development state of the project at the moment when the preliminary report was delivered to the board members and the planning until finishing all its tasks. This planification was done the first week of March, as we can see in the shaded column in the chronogram.

We can observe yellow coloured tasks representing such tasks that are not achieved a third of its work charge whereas blue coloured tasks represent task with are planified but their start date is greater that actual date. Finally, green coloured tasks indicate that their work has been done and they are finished.

It is important to note that at that time the dates for the administrative tasks were provisional and has been changed according to the decisions and posiblities made after this chronogram was done.

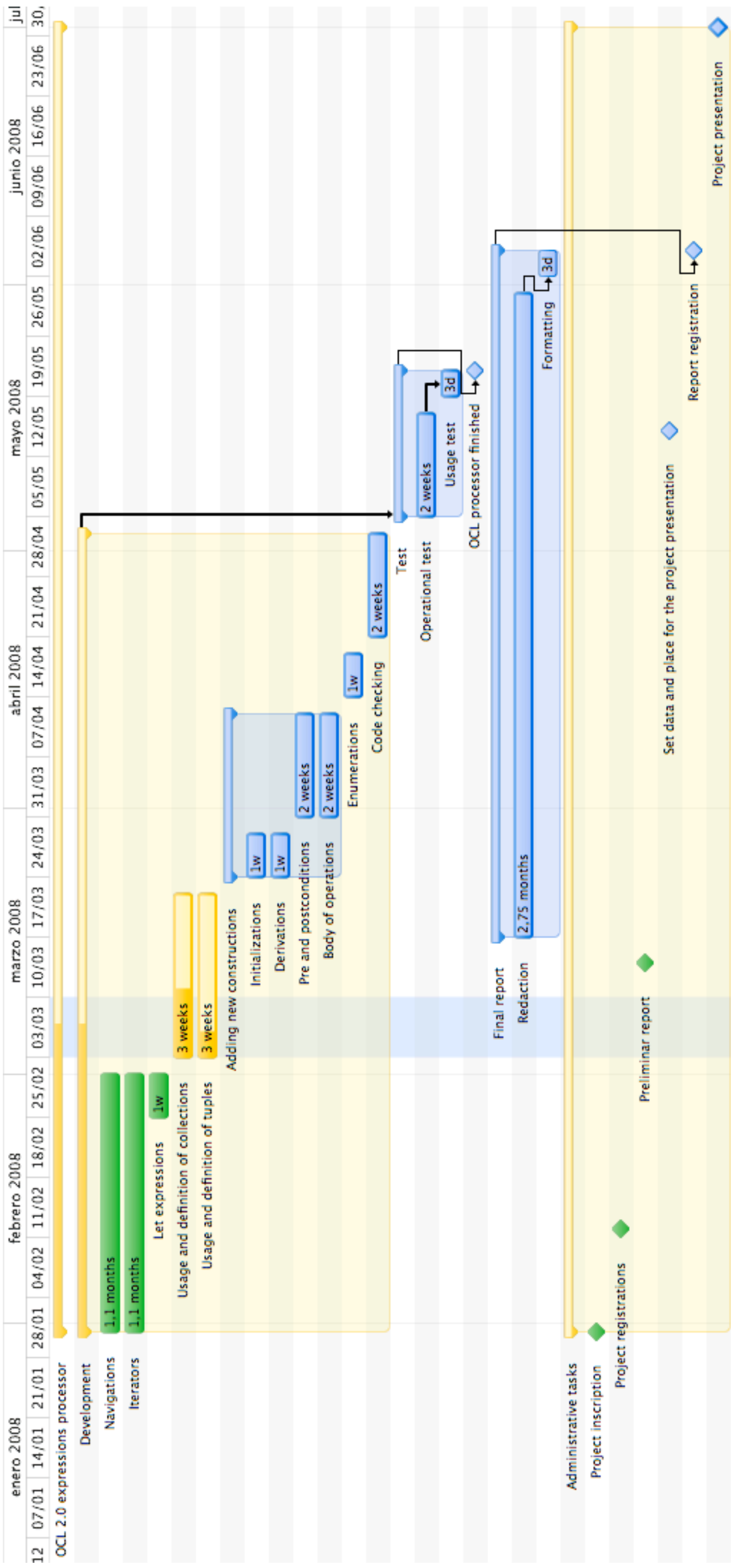


Figure 18.1: Estimated plan

18.3 REAL PLAN

In this section we will show the real planification after finishing this final career project with all changes made to the estimated planification.

In this new chronogram we can observe green coloured tasks representing tasks that have been finished. We can also see that the milestone referencing the date of the project presentation is marked in blue because the date of this final career project is previous to the presentation date, therefore such task is planified although its start date has not arrived yet.

It is important to emphasize that in the chronogram some shaded areas appear behind the representation of finished tasks in order to denote the estimated times planified in the previous chornogram showed in the Figure 18.1.

We can see delays in the planification tasks of the development phase and the tests due to some problems that made me dedicate less time that needed to do the necessary work. Anyways, such delays had not affected hardly the initial planning because now we are able to affirm that our processor for OCL 2.0 expressions is finished in its first complete release version and it is ready to be used for those that are interested in our work.

Finally we can say that the estimated planification was a good guideline to follow in order to achieve the deadlines of all tasks. Furthermore, due to the time dedicated before the formally start of the project to begin it we had sufficient security to feel that we would be able to finish the plan although some problems happened.

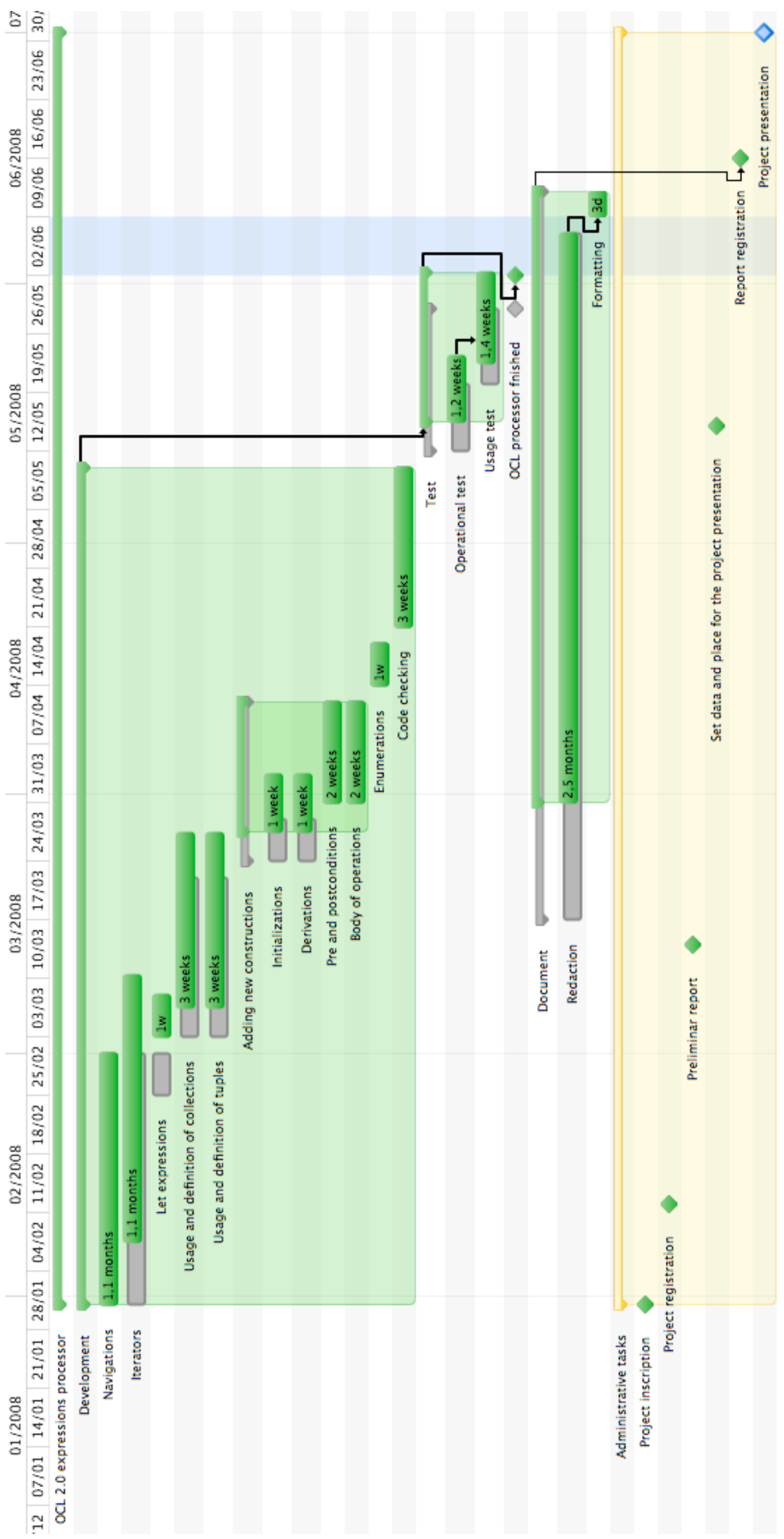


Figure 18.2: Real plan

COST 19

19 COST

19.1 INTRODUCTION

To have an idea about the economical cost of the project we will show a little study about the work dedication and the workers needed to do it.

At next sections we will introduce an approximate evaluation of the different activities that conform this project including the computation of the number of hours neede to complete them.

Then, with this number of hours we will be able to estimate the total cost of the project by means of the average of salary that the participant workers could obtain for doing their work.

19.2 WORK DEDICATION

First of all, we must note that our project can be divided into two parts according to two different skills of workers. We will need the support of an analyst that will work in the activities shown in the Table 19.1. In this table we can see every activity with its duration in days and the dedication of the worker. Finally, we can calculate the number of hours for each activity assuming that each day consists on a period of eight hours.

For example, first activity takes five days that implies 40 hours (5 x 8), but the dedication is not complete (80%) so finally we have 40 x 80% = 32 hours.

Activity	Days	Dedication	Hours
Analize existent OCL tools	5	80%	32
Analize the requirements of our system	5	60%	24
Choose a tool to be the basis of our processor	4	70%	22,4
Analize SableCC and SableCC-Ext	5	80%	32
Adapt OCL 2.0 grammar	15	30%	36
Conceptual schema of generated classes	4	60%	19,2
Analize future extensions and improvements	3	40%	9,6
Final report	50	70%	280
		TOTAL	**455,2**

Table 19.1: Work dedication for an analyst

Activity	Days	Dedication	Hours
Study Eina GMC	5	50%	20
Study Dresden tool	5	50%	20
Generate SableCC-Ext independent release package adapted to Eina GMC	5	70%	28
Develop constraints instantiation	100	80%	640
Develop type checking	40	40%	128
Develop facade methods	4	30%	9,6
Develop delete visitor	10	40%	32
Develop inverse converter	15	30%	36
Tests	5	60%	24
		TOTAL	**937,6**

Table 19.2: Work dedication for a programmer

On the other hand, we will also need a programmer to develop the processor according to the indications of the analyst. In the Table 19.2 we have all the activities with their needed hours that have been done by the programmer.

19.3 ECONOMIC REPORT

Once we have the number of hours for both the analyst and the programmer, we only have to know how much have to pay to every one according to its salaries.

At [IJTw] we can obtain an approximate idea about the salary trends for analysts and programmers. In the next figure extracted from that web we can see in red colour the average of the analyst's salary and in green colour the average of the programmer's salary.

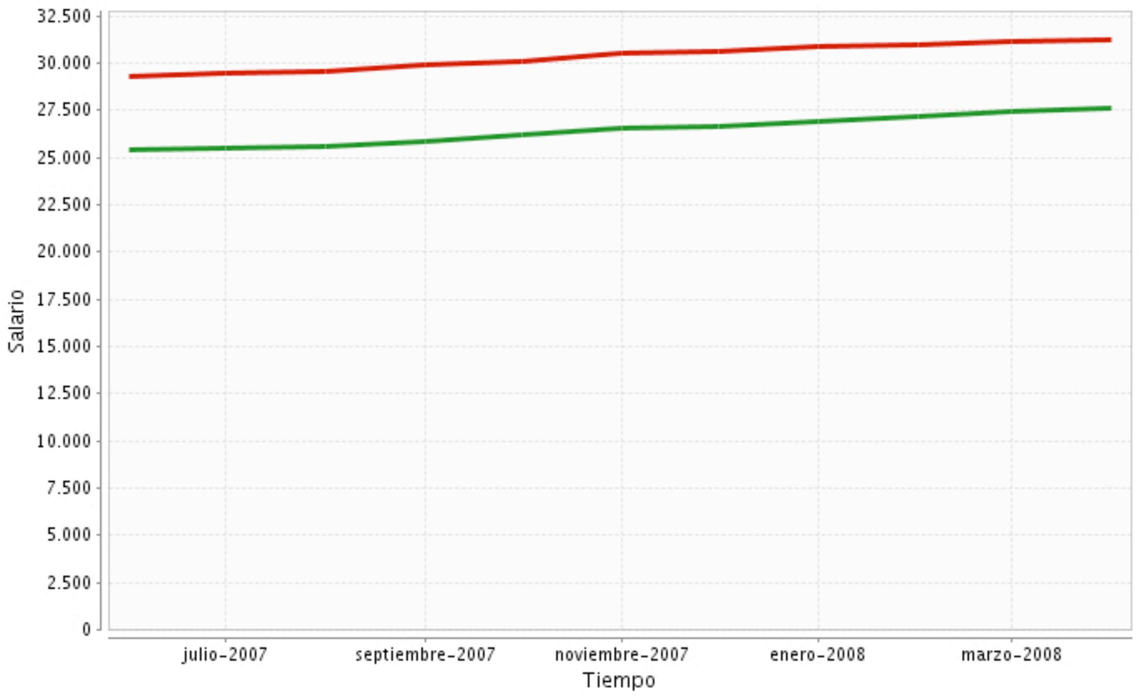


Figure 19.1: Comparison between salary evolutions of analyst versus programmer from July 2007 to March 2008 extracted from [IJTw]

With this information we can decide that to calculate the cost of the project we choose an approximate value of 30.000€ as the salary of an analyst per year, and a value of 26.000€ as the salary of a programmer per year.

If we suppose that these amounts can be divided in 14 payments, and a month consists on four weeks of five workable days, each one with eight hours, we should divide the initial amounts by 14 x 4 x 5 x 8 = 2240. If we do it, we will obtain the cost per hour for each of these workers.

Worker	Hours	Salary (€ / hour)	Total
Analyst	455,2	13,40	6.099,68€
Programmer	937,6	11,60	10.876,16€
		TOTAL	**16.975,84€**

Table 19.3: Final cost of the project

Finally, in the Table 19.3 we find such computations and the final cost of the project that mixes the salaries of both workers. Therefore the approach to the economical cost of our project consists on an approximate value of 16.975,84€.

CONCLUSIONS 20

20 CONCLUSIONS

20.1 ABOUT THE FINAL CAREER PROJECT

- Final career project is the last activity in the university studies. It should be a project in which we have to demonstrate the knowledge acquired during the previous years in order to obtain the final diploma.

- Students should choose a topic to investige that could motivate them in order to obtain better results.

- My preferences about compilers and software engineering made me choose this topic, in which I am able to mix these two study areas into the construction of a processor of OCL 2.0 expressions.

20.2 ABOUT CONCEPTUAL MODELING

- Conceptual modeling is an important activity in software engineering. Its aim is to obtain a conceptual schema for a software system.

- Conceptual schemas contain all the information needed of a software system. Thet are useful to support the different phases of the software development cycle.

- All participants in a software development project have to collaborate in order to create a correct conceptual schema. This union improves communication between different members and helps finding problems in earlier stages of the project construction.

- Although it could not be written in a physical document, all analysts, engineers, programmers and more have their own vision of the conceptual schema of the system. Therefore, to write such ideas in a common document is a solution for to avoid different points of view.

20.3 ABOUT MODELING LANGUAGES

- Both UML and OCL are essential languages to use inside the conceptual modeling area of software engineering.

- UML is useful to construct the conceptual schema of a software system. Nevertheless, it needs the power of the OCL to achieve a complete specification due to the existent lacks of the UML covered by the OCL.

- Both languages are the basis of the Model Driven Architecture that will be the future of the software development. MDA could be a new abstraction layer upon the actual abstraction provided by the high-level programming languages.

- XMI language provides the possibility of sharing models in an standard way, avoiding errors where information is exchanged between different modeling tools.

20.4 ABOUT MODELING TOOLS

- There exist a lot of modeling tools to be used in order to simplify the conceptual modeling activity.

- We have studied four of these tools to decide wich one could be the chosen tool to be the basis of our development. It is important to note that nowadays to start a new project from scratch is not a good idea. Therefore, we preferred to base our processor in an existent desing.

- Before studying and evaluating a set of four tools containing the USE tool, the Dresden OCL2 Toolkit, the MOVA tool and the IBM OCL Parser, we decided to base our implementation in the parser subsystem of the Dresden OCL2 Toolkit due to its architecture and features were more similar to ours than others.

20.5 ABOUT THE PROCESSOR OF OCL 2.0 EXPRESSIONS

- Before our work, Eina GMC has a lack with the Object Constraint Language. After the implementation of our processor, we can affirm that such lack has been erased and now the bases of Eina GMC are completed.

- The processor of OCL 2.0 expressions provides Eina GMC with the processing of OCL expressions, their instantiation into the UML and OCL metamodels, and a correctness checking phase that feedbacks users if some errors are found into such expressions.

- Furthermore, our processor provides the inverse conversion from metamodel instances to original textual expressions. With all of it the OCL checking process is finished.

- Our processor can be used in different ways according to the experience of the users: directly through Java code calling the processor by means of facade methods or constructing the processor components and making a main program, opening a demo GUI in wich load XMI models and parse OCL expressions, or even writing OCL constraints directly inside comments of the Poseidon for UML CASE tool.

- We have followed a modular design for our processor that simplifies the changeability and makes easier to adapt it to future new versions of UML or OCL.

20.6 ABOUT IMPROVEMENTS AND EXTENSIONS

- Our processor needs to be completed with secondary constructions like named self or message expressions that although are not essential, to support them is a needed task.

- Furthermore, the possibility of support user data types will be added once we decide how to solve the problem that such elements introduce to the type checking process.

- Finally, we have to join all the components of the Eina GMC in order to construct a main GUI program in wich all users could get in touch with our implementations in an easier way. Once we have all these components completed to develop such program is only a time problem.

20.7 ABOUT THE ACQUIRED EXPERIENCE

- After finishing this document and therefore this final career project it is important to emphasize that all different kinds of knowledge acquired during the career classes, theoretical and practical, have been essential to start and finish the project.

- Furthermore, during the development of the project I have acquired extra knowledge about the deeply studied areas. In addition, the realization of a final career project implies a high activity doing self-study and acquiring this new knowledge by myself. It can be said that during this phase I have been my own teacher.

- Finally, I have to say that I have obtained more than knowledge. Now I have more security of being able to make important developments and to manage complex projects.

GLOSSARY

A

A GLOSSARY

API
Abbreviation of application programming interface, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing documentation and examples of usage related to its libraries and structures.

Automaton
A graph structure made by states and edges that represents a language. It processes words and indicates if each processed word belongs or not to the represented language.

Command Line Interface (CLI)
A mechanism for interacting with a computer operating system or software by typing commands to conduct the system.

Compiler
Software tool that translates code written in a language into another language.

Compilers compiler
Software tool that generates the source code for a parser, interpreter, or compiler from a language description.

Context
The element specified in an UML model for which an OCL 2.0 expression is defined.

Constraint
An expression that restricts some characteristics, functionalities or behaviour of an element.

Derivation rule
An expression that indicates how to compute the value of a derived element.

Diagram
A schematic drawing showing the relation between the elements of a *model*.

Graphical User Interface (GUI)
User interface that allows people to interact with a software system through direct manipulation of graphical icons, visual indicators or special graphical elements with a mouse peripheral.

Inherited attributes
Inherited attributes help pass semantic information down the parse tree from parents to its descendants.

Interpreter
Software tool that executes code written in a language.

Invariant
A boolean expression that must be carry out for the element where it is defined.

Iterator
An object which allows an user to traverse through all the elements of a collection.

Iterator variable
A variable that is used within the body expression of a loop expression like an iterator or the iterate construction. It has the value of the source collection elements in which the loop expression is applied.

Language
A set of characters and symbols and syntactic rules for their combination and use in a communication process.

Metaclass
A class belonging to a *metamodel*.

Metadata repository
A database of data about data. It provides a consistent and reliable means of access to data.

Metamodel
A *model* of models. It can also be defined as a model describing how to define models.

Model
A simplified, consistent, unambiguous and coherent description of a complex system or process. It contains model elements with features and restrictions.

Model Driven Architecture
Software development method that consists in the automated transformation of *platform-independent models* into *platform-specific models*.

Navigation
The action of access to different elements from the contextual element through associations.

Parser
One of the components in an interpreter or compiler, where it captures the implied hierarchy of the input text and transforms it into a form suitable for further processing (often some kind of parse tree, abstract syntax tree or other hierarchical structure) and normally checks for syntax errors at the same time.

Platform-independent model (PIM)
A model of a software or business system that is independent of the specific technological platform used to implement it

Platform-specific model (PSM)
A model of a software or business system that is linked to a specific technological platform (e.g. a specific programming language, operating system or database).

Postcondition
A condition or predicate that must always be true just after the execution of some section of code.

Precondition

A condition or predicate that must always be true just prior to the execution of some section of code

Query operation
An operation without side effects. It can also be known as a read-only operation.

Semantics
The study of meaning.

Symbol table
A data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

Syntactic sugar
Syntax of a computer language that do not affect its functionality but make it "sweeter" or easier for humans to use.

Syntax
The grammatical rules and structural patterns governing the ordered use of appropriate words and symbols in a particular language.

Synthesized attribute
The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes. Synthesized attributes are used to pass semantic information up the parse tree

System
Set of entities, real or abstract, comprising a whole.

Type
A term that indicates a class, a data type or each subclass from the Type *metaclass* of the UML and OCL 2.0 metamodels.

UML
A standardized visual specification language for object modeling.

OCL 2.0 GRAMMAR B

B OCL 2.0 GRAMMAR

In this appendix we show the complete grammar for the OCL 2.0 language used within our processor of expressions. Note that it is written using SableCC Ext syntax.

```
Package parser.sablecc;

Helpers
  all                    = [0 .. 0xffff];
  lf                     = 10;
  cr                     = 13;
  tab                    =  9;
  space                  = ' ';
  enye                   = 241;
  line_terminator        = cr | lf | cr lf;
  input_character        = [all - [cr + lf]];
  decimal_digit          = ['0' .. '9'];
  octal_digit            = ['0' .. '7'];
  hex_digit              = ['0' .. '9'] | ['a' .. 'f'] | ['A' .. 'F'] ;
  loweralpha             = [['a' .. 'z']+ enye];
  upperalpha             = ['A' .. 'Z'];
  name_start_character   = [loweralpha + upperalpha];
  name_character         = [[name_start_character + '_'] + decimal_digit];
  char_e                 = 'e' | 'E';
  sign                   = '+' | '-';
  backslash              = '\';
  quote                  = '"';
  tick                   = ''';
  dot                    = '.';
  minus                  = '-';
  dbl_dash               = '--';
  commentblock_start     = '/*';
  commentblock_end       = '*/';
  noasterisk             = [ all - '*' ];
  noslash                = [ all - '/' ];
  asterisk               = '*';
  string_not_unescaped   = [ ''' + [ backslash + [cr + lf] ] ];
  basic_escape_code      = 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' | tick
                         | quote | '?' | backslash ;

octal_escape_code        = octal_digit ( octal_digit octal_digit? )? ;
  hex_escape_code        = 'x' hex_digit ( hex_digit (hex_digit hex_digit?
                                        )? )? ;
  escape_sequence        = backslash ( basic_escape_code |
                         octal_escape_code | hex_escape_code );
  string_literal_part    = [all - string_not_unescaped] | escape_sequence;
```

```
Tokens
! newline                = line_terminator;
! blank                  = tab | space* ;
! commentline            = dbl_dash input_character* line_terminator;
! commentblock           = commentblock_start ( noasterisk | asterisk
                           noslash )* commentblock_end;
! tick                   = ''';
! paren_open             = '(';
! paren_close            = ')';
! comma                  = ',';
! arrow_right            = '->';
! dot                    = '.';
! dbl_dot                = '..';
! colon                  = ':';
! dbl_colon              = '::';
! semi_colon             = ';';
  equals                 = '=';
! question_mark          = '?';
! hash                   = '#';
! at_pre                 = '@pre';
! bag                    = 'Bag';
! collection             = 'Collection';
! ordered_set            = 'OrderedSet';
! sequence               = 'Sequence';
! set                    = 'Set';
! tuple                  = 'Tuple';
! tuple_type             = 'TupleType';
! bracket_open           = '[';
! bracket_close          = ']';
! body                   = 'body';
! context                = 'context';
! def                    = 'def';
! derive                 = 'derive';
! else                   = 'else';
! endif                  = 'endif';
! endpackage             = 'endpackage';
  false <Boolean>        = 'false';
! if                     = 'if';
! in                     = 'in';
! init                   = 'init';
! inv                    = 'inv';
! let                    = 'let';
! package                = 'package';
! pre                    = 'pre';
! post                   = 'post';
! then                   = 'then';
  true <Boolean>         = 'true';
! brace_open             = '{';
! brace_close            = '}';
! vertical_bar           = '|';
  integer_literal <Integer>   = decimal_digit+;
  real_literal <Double>       = decimal_digit+ dot decimal_digit+
                              | decimal_digit+ dot decimal_digit+ char_e
                              sign? decimal_digit+;
  string_literal <String>     = tick string_literal_part* tick;
  iterate                = 'iterate';
  select                 = 'select';
  reject                 = 'reject';
  collect                = 'collect';
  for_all                = 'forAll';
  any                    = 'any';
  exists                 = 'exists';
  one                    = 'one';
  is_unique              = 'isUnique';
  sorted_by              = 'sortedBy';
  collect_nested         = 'collectNested';
  ocl_op_is_type_of      = 'oclIsTypeOf';
  ocl_op_is_kind_of      = 'oclIsKindOf';
  ocl_op_as_type         = 'oclAsType';
  minus                  = '-';
  star                   = '*';
  slash                  = '/';
  plus                   = '+';
  rel_gt                 = '>';
  rel_lt                 = '<';
  rel_gte                = '>=';
  rel_lte                = '<=';
  rel_notequal           = '<>';
  log_and                = 'and';
  log_or                 = 'or';
  log_xor                = 'xor';
  log_implies            = 'implies';
  not                    = 'not';
  simple_name            = name_start_character name_character*;

Ignored Tokens
  commentline, commentblock, newline, blank;

Productions
  context_declaration_list_cs <List> =
        [context]:context_declaration_cs
        [tail]:context_declaration_list_cs? #nocreate
    ;

  context_declaration_cs <OclContextDeclaration> =
    {classifier} <OclClassifierContextDecl>
                      context [context_name]:path_name_cs
                      [constraints]:classifier_constraint_cs+
                      #customheritage
    | {attr_or_assoc} <OclAttrOrAssocContextDecl>
                      context [context_name]:path_name_cs
                      colon [type]:type_specifier
                      [constraints]:init_or_der_value_cs+
                      #customheritage
    | {operation}   <OclOperationContextDecl>
                      context [context_name]:path_name_cs
                      [signature]:operation_signature_cs
                      [constraints]:operation_constraint_cs+
                      #customheritage
    ;

   classifier_constraint_cs <OclClassifierConstraint> =
     {invariant} <OclInvariantClassifierConstraint>
```

```
        inv [name]:simple_name? colon [invariant]:ocl_expression_cs
     |{definition} <OclDefinitionClassifierConstraint>
       def [name]:simple_name? colon definition]:definition_constraint_cs
     ;

    definition_constraint_cs <OclDefinitionConstraint> =
         [entity]:defined_entity_decl_cs
         !equals
         [definition]:ocl_expression_cs #customheritage
     ;

    defined_entity_decl_cs <OclDefinedEntityDecl> =
      {attribute} <OclAttributeDefinedEntityDecl>
                         [attribute]:formal_parameter_cs
     |{operation} <OclOperationDefinedEntityDecl>
                         [operation_name]:simple_name
                         [operation]:operation_signature_cs
     ;

    operation_signature_cs <OclOperationSignature> =
         paren_open parameters]:formal_parameter_list_cs?
         paren_close [return_type]:operation_return_type_specifier_cs?
     ;

    operation_return_type_specifier_cs <Classifier> =
         colon [return_type]:type_specifier #chain
     ;

    operation_constraint_cs <OclOperationConstraint> =
         [stereotype]:op_constraint_stereotype_cs
         [name]:simple_name? colon
         [expression]:ocl_expression_cs #customheritage
     ;

    op_constraint_stereotype_cs <OclOperationConstraintStereotype> =
         {pre} pre   #nocreate | {post} post #nocreate
     |   {body} body #nocreate
     ;

    init_or_der_value_cs <OclAttrOrAssocConstraint> =
         {init} <OclInitConstraint>
          init [name]:simple_name? colon [initializer]:ocl_expression_cs
     |   {derive} <OclDeriveConstraint>
          derive [name]:simple_name? colon
          [derive_expression]:ocl_expression_cs
     ;

   identifier_cs <String> =
       {simple} simple_name #chain
     | {iterate} iterate #chain
     | {iterator_name} iterator_name_cs #chain
     | {ocl_op_name} ocl_op_name #chain
     ;

   path_name_cs <List> = [qualifier]:path_name_head_cs*
                          [name]:identifier_cs #nocreate;
```

```
path_name_head_cs <String> = identifier_cs dbl_colon #chain;

iterator_name_cs <String> =
    {select} T.select #chain    | {reject} T.reject #chain
  | {collect} T.collect #chain | {for_all} T.for_all #chain
  | {exists} T.exists #chain    | {any} T.any #chain
  | {one} T.one #chain          | {is_unique} T.is_unique #chain
  | {sorted_by} T.sorted_by #chain
  | {collect_nested} T.collect_nested #chain
  ;

 ocl_op_name <String> =
    {kind_of} T.ocl_op_is_kind_of      #chain
  | {type_of} T.ocl_op_is_type_of      #chain
  | {as_type} T.ocl_op_as_type         #chain
  ;

ocl_expression_cs <OclExpression> =
    {with_let}    let_exp_cs      #chain
  | {without_let} logical_exp_cs  #chain
  ;

let_exp_cs <LetExp> =
      let [variables]:initialized_variable_list_cs
      in  [expression]:expression #customheritage #nocreate
  ;

if_exp_cs <IfExp> =
   if [condition]:logical_exp_cs
      then [then_branch]:ocl_expression_cs
      else [else_branch]:ocl_expression_cs
   endif
  ;

logical_exp_cs <OclExpression> =
  {chain}    <OclExpression>    [operand]:relational_exp_cs #chain
  | {binary} <OperationCallExp> [operand]:relational_exp_cs
             [tail]:logical_exp_tail_cs+
  ;

logical_exp_tail_cs <OclBinaryExpTail> =
    [operator]:logic_op [operand]:relational_exp_cs
  ;

logic_op <String> =
    {and} log_and #chain  | {or} log_or            #chain
  | {xor} log_xor #chai   | {implies} log_implies  #chain
  ;

relational_exp_cs <OclExpression> =
  {chain} [operand]:additive_exp_cs #chain
  |{binary} <OperationCallExp> [operand]:additive_exp_cs
                               [tail]:relational_exp_tail_cs
  ;

relational_exp_tail_cs <OclBinaryExpTail> =
      [operator]:rel_op [operand]:additive_exp_cs
  ;

rel_op <String> =
    {eq}  equals  #chain   | {ne} rel_notequal #chain
  | {gt}  rel_gt  #chain   | {lt} rel_lt       #chain
  | {gte} rel_gte #chain   | {lte} rel_lte     #chain
  ;

additive_exp_cs <OclExpression> =
    {chain} [operand]:multiplicative_exp_cs #chain
  | {binary} <OperationCallExp>  [operand]:multiplicative_exp_cs
                                [tail]:additive_exp_tail_cs+
  ;

additive_exp_tail_cs <OclBinaryExpTail> =
      [operator]:add_op [operand]:multiplicative_exp_cs
  ;

add_op <String> =
    {plus} plus #chain | {minus} minus #chain
  ;

multiplicative_exp_cs <OclExpression> =
    {chain} [operand]:unary_exp_cs #chain
  | {binary} <OperationCallExp> [operand]:unary_exp_cs
                               [tail]:multiplicative_exp_tail_cs+
  ;

multiplicative_exp_tail_cs <OclBinaryExpTail> =
      [operator]:mult_op [operand]:unary_exp_cs
  ;

mult_op <String> =
    {mult} star #chain | {div} slash #chain
  ;

unary_exp_cs <OclExpression> =
    {unary_op} <OperationCallExp> [operator]:unary_op
                                 [operand]:postfix_exp_cs
  | {unary_nop} [postfix]:postfix_exp_cs  #chain
  ;

unary_op <String> =
    {minus} minus #chain | {not} not #chain
  ;

postfix_exp_cs <OclExpression> =
    {primary}   [primary]:primary_exp_cs #chain
  | {with_tail} [leftmost_exp]:primary_exp_cs postfix_exp_tail_cs+
                #maketree #nocreate
  ;

postfix_exp_tail_cs <OclExpression> =
  {prop} dot [prop_call]:property_call_exp_cs #customheritage #nocreate
  | {arrow_prop} arrow_right [tail]:arrow_property_call_exp_cs #chain
  ;
```

```
primary_exp_cs <OclExpression> =
    {literal} literal_exp_cs #chain
  |{parenthesized} paren_open expression paren_close #chain
  | {property} [prop_call]:property_call_exp_cs #customheritage
    #nocreate
  |{if} if_exp_cs #chain
  ;

expression <OclExpression> = ocl_expression_cs #chain;

property_call_exp_cs <OclExpression> =
  {path_time} <OclExpression> [name]:path_name_cs [time]:time_exp_cs?
    #nocreate
  |{arg_list} <OclExpression> [name]:path_name_cs  [time]:time_exp_cs?
    [parameters]:property_call_parameters_cs #customheritage #nocreate
  |{qualified} <OclExpression> [name]:path_name_cs
    [qualifiers]:qualifiers #customheritage [time]:time_exp_cs?
    #nocreate
  ;

property_call_parameters_cs <List> =
    paren_open [param_list]:actual_parameter_list_cs? #customheritage
    paren_close #nocreate
  ;

qualifiers <List> =
    bracket_open qualifiers_list_cs #customheritage bracket_close
    #chain
  ;

qualifiers_list_cs <List> =
   [element]:qualifiers_list_element_cs [tail]:qualifiers_list_tail_cs?
   #nocreate
  ;

qualifiers_list_element_cs <String> =
   {qualifier}simple_name #chain
  ;

qualifiers_list_tail_cs <List> =
   comma [tail]:qualifiers_list_cs #chain
  ;

arrow_property_call_exp_cs <OclExpression> =
   {iterate} <IterateExp>
    T.iterate paren_open [iterators]:iterate_vars_cs? #customheritage
       [accumulator]:initialized_variable_cs vertical_bar
       [body]:expression #customheritage paren_close
  | {iterator} <IteratorExp>
       [name]:iterator_name_cs paren_open
       [iterators]:iterator_vars_cs? #customheritage
       [body]:expression #customheritage paren_close
  | {operation} <OperationCallExp>
       [name]:simple_name paren_open
       [parameters]:actual_parameter_list_cs?
       #customheritage paren_close
  ;
```

```
iterate_vars_cs <List> =
    [iterators]:actual_parameter_list_cs #customheritage
    semi_colon #chain
  ;

iterator_vars_cs <List> =
    [iterators]:actual_parameter_list_cs #customheritage
    vertical_bar #chain
  ;

initialized_variable_list_cs <List> =
      [item]:initialized_variable_cs
      [tail]:initialized_variable_list_tail_cs* #nocreate
  ;

initialized_variable_list_tail_cs <Variable> =
      comma [item]:initialized_variable_cs #chain
  ;

initialized_variable_cs <Variable> =
      [name_and_type]:formal_parameter_cs
      [initializer]:variable_initializer
  ;

variable_initializer <OclExpression> =
      !equals [init_value]:ocl_expression_cs #chain
  ;

actual_parameter_list_cs <List> =
    [element]:actual_parameter_list_element_cs
    [tail]:actual_parameter_list_tail_cs? #nocreate
  ;

actual_parameter_list_tail_cs <List> =
      comma [tail]:actual_parameter_list_cs #chain
  ;

actual_parameter_list_element_cs <OclActualParameterListItem> =
    {untyped} [element]:expression
  | {typed}   [param]:formal_parameter_cs
  ;

formal_parameter_cs <OclFormalParameter> =
      [name]:simple_name [type]:formal_parameter_type_specifier
  ;

formal_parameter_type_specifier <Classifier> =
      colon [type]:type_specifier #chain
  ;

formal_parameter_list_cs <List> =
      [item]:formal_parameter_cs
      [tail]:formal_parameter_list_tail_cs ? #nocreate
  ;

formal_parameter_list_tail_cs <List> =
      comma [tail]:formal_parameter_list_cs #chain
  ;

type_specifier <Classifier> =
   {simple_type} simple_type_specifier_cs #chain #nocreate
 | {collection_type} collection_type_specifier_cs #chain #nocreate
 | {tuple_type} tuple_type_specifier_cs #chain #nocreate
  ;

simple_type_specifier_cs <Classifier> =
      path_name_cs #nocreate
  ;

collection_type_specifier_cs <CollectionType> =
      [kind]:collection_type_identifier_cs paren_open
      [type]:type_specifier paren_close #nocreate
 ;

collection_type_identifier_cs <CollectionKind> =
   {set} set #nocreate            | {bag} bag #nocreate
 | {sequence} sequence #nocreate | {collection} collection #nocreate
 | {ordered_set} ordered_set #nocreate
 ;

tuple_type_specifier_cs <TupleType> =
      tuple_type paren_open [tuple_members]:formal_parameter_list_cs?
      paren_close #nocreate
  ;

time_exp_cs <OclTimeExp> =
      is_marked_pre_cs #chain
  ;

is_marked_pre_cs <OclTimeExp> =
      at_pre #nocreate
  ;

literal_exp_cs <LiteralExp> =
   {lit_primitive}    primitive_literal_exp_cs    #chain
 | {lit_collection}   collection_literal_exp_cs   #chain
 | {lit_tuple}        tuple_literal_exp_cs        #chain
 ;

primitive_literal_exp_cs <PrimitiveLiteralExp> =
    {numeric} numeric_literal_exp_cs  #chain
  | {string} string_literal_exp_cs    #chain
  | {boolean} boolean_literal_exp_cs  #chain
  ;

numeric_literal_exp_cs <NumericLiteralExp> =
    {int}  <IntegerLiteralExp> [integer]:integer_literal
  | {real} <RealLiteralExp>    [real]:real_literal
  ;

string_literal_exp_cs <StringLiteralExp> =
      [value]:string_literal
  ;
```

```
boolean_literal_exp_cs <BooleanLiteralExp> =
    {false} false #nocreate | {true} true #nocreate
  ;

tuple_literal_exp_cs <TupleLiteralExp> =
      tuple
      brace_open
      [tuple_part]:tuple_part_list_cs
      brace_close
  ;

tuple_part_list_cs <List> =
      [item]:tuple_part_cs
      [tail]:tuple_part_list_tail_cs* #nocreate
  ;

tuple_part_list_tail_cs <Variable> =
      comma
      [item]:tuple_part_cs #chain
  ;

tuple_part_cs <Variable> =
      [name]:simple_name
      tuple_part_type?
      [initializer]:variable_initializer
  ;

tuple_part_type <Classifier> =
      !colon [type]:type_specifier #chain
  ;

collection_literal_exp_cs <CollectionLiteralExp> =
      [kind]:collection_type_identifier_cs
      brace_open
      [parts]:collection_literal_parts_cs?
      brace_close
  ;

collection_literal_parts_cs <List> =
      [part]:collection_literal_part_cs
      [tail]:collection_literal_parts_tail_cs? #nocreate
  ;

collection_literal_parts_tail_cs <List> =
      comma [tail]:collection_literal_parts_cs #chain
  ;

collection_literal_part_cs <CollectionLiteralPart> =
      {range}      collection_range_cs #chain
  |   {single_exp} <CollectionItem> expression
  ;

collection_range_cs <CollectionRange> =
      [first]:expression dbl_dot [last]:expression
  ;
```

BIBLIOGRAPHY C

C BIBLIOGRAPHY

[Dau04] Berthold Daum. *Professional Eclipse 3 for Java Developers.* Wrox, November 2004, 1st Edition.

[CM03] Cases, R. i Màrquez, L. *Llenguatges, gramàtiques i autòmats. Curs basic.* Edicions UPC, 2003.

[DBLPw] Universität Trier. *Digital Bibliography & Library Project* (Web Site). http://www.informatik.uni-trier.de/~ley/db/

[DOTw] Dresden University of Technology, Department of Computer Science. *Dresden OCL2 Toolkit* (Welcome Page). http://dresden-ocl.sourceforge.net

[ECLw] Eclipse Community. *Eclipse Integrated Development Environment.* (Home Page) http://www.eclipse.org

[EINw] GMC: Research Group in Conceptual Modeling of Information Systems. *Eina GMC* (Eina GMC Project Page). http://guifre.lsi.upc.edu/eina_GMC/index.html

[Gag98] Étienne Gagnon. *SABLECC, an object-oriented compiler framework. Master thesis.* School of Computer Science, McGill University, Montreal, 1998.

[GDB02] Timothy J. Grose, Gary C. Doney and Stephen A. Brodsky. *Mastering XMI: Java Programming with XMI, XML, and UML.* Wiley Computer Publishing, April 2002.

[GH95] E. Gamma and R. Helm. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[GMC07] GMC: Research Group in Conceptual Modeling of Information Systems. *Eina GMC Start Guide*. 2007. http://guifre.lsi.upc.edu/eina_GMC/

[GMCw] GMC: Research Group in Conceptual Modeling of Information Systems. *GMC* (Home Page). http://guifre.lsi.upc.edu/index.html

[HKR02] Hammond J., Keeney R. and Raiffa H. *Smart Choices: A Practical Guide to Making Better Decisions*. Broadway, 2002

[HMU01] Hopcroft, J.E.; Motwani, R. i Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001. 2nd Edition.

[IJTw] InfoJobs Trends Salarios. *Información salarial* (Web Site) http://salarios.infojobs.net/index.cfm

[IOPw] Jos Warmer. *IBM OCL Parser v.0.3.* 1997 (Web Site) http://www-306.ibm.com/software/awdtools/library/standards/ocl-download.html

[JRAw] The Java Tutorials. *The Java Reflection API* (Web Page) http://java.sun.com/docs/books/tutorial/reflect/index.html

[Kon05] Ansgar Konermann. *The Parser Subsystem of the Dresden OCL2 Toolkit: Design and Implementation.* September 2005. http://dresden-ocl.sourceforge.net/papers/ParserDesign.pdf

[MDAw] Object Management Group. *Model-Driven Architecture* (Web Site) http://www.omg.org/mda

[MDRw] NetBeans Community. *Metadata Repository Project* (Project Home page). http://mdr.netbeans.org/

[MOVw] Modeling and Validation (MOVA) Group. *The MOVA Project Home* (Home Page). http://maude.sip.ucm.es/mova/

[NETw] NetBeans Community. *NetBeans Integrated Development Environment.* (Home Page) http://www.netbeans.org

[Obj02] Object Management Group. *Meta Object Facility (MOF) Spefication, Version 1.4,* April 2002 http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[Obj06] Object Management Group. *Object Constraint Language – OMG Available Specification, Version 2.0,* May 2006. http://www.omg.org/cgi-bin/doc?ptc/06-05-01.

[Obj07] Object Management Group. *Unified Modeling Language: Superstructure – OMG Available Specification, Version 2.1.2,* November 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.

[Obj07x] Object Management Group. *XML Metadate Interchange – OMG Available Specification, Version 2.1,* December 2007. http://www.omg.org/spec/XMI/2.1.1/

[Oli07] Antoni Olivé. *Conceptual Modeling of Information Systems,* 2007. Springer.

[PDGPE]	Notes about PDGPE subject. *Presa de decisions i gestió de projectes empresarials.* Departament d'Organització d'Empreses. Computer Science studies of Barcelona School of Informatics. http://www.fib.upc.edu/en/infoAca/estudis/assignatures/PDGPE
[PFUw]	Gentleware. *Poseidon for UML, Version 6.0.* (Web Site). http://www.gentleware.com/
[PO06]	Elena Planas, Antoni Olivé. *The DBLP Case Study,* 2006. http://guifre.lsi.upc.edu/CaseStudies.html
[RJB04]	James Rumbaugh, Ivar Jacobson ad Grady Booch. *The Unified Modeling Language: Reference Manual, Second Edition.* July 2004. Addison-Wesley
[SCCw]	Étienne Gagnon. *SableCC parser generator* (Home Page). http://sablecc.org
[UOCw]	Open University of Catalonia. *UOC* (Web Site). http://www.uoc.edu
[UPCw]	Technical University of Catalonia. *UPC* (Web Site). http://www.upc.edu/
[USE07]	Database Systems Group, Bremen University. *USE, A UML based Specification Environment, Preliminary Version 0.1,* May 2007. http://www.db.informatik.uni-bremen.de/projects/USE/use-documentation.pdf
[USEw]	Database Systems Group, Bremen University. *USE Project* (Web Site) http://www.db.informatik.uni-bremen.de/projects/USE/
[Villw]	Villegas Niño, Antonio. *Persona Web.* http://www.lsi.upc.edu/~avillegas
[WK03]	J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 2003. 2nd Edition.
[WM95]	R. Wilhelm, D. Maurer. *Compiler Design: Theory, Constuction, Generation.* Addison-Wesley, 1995.
[XMLw]	World Wide Web Consortium (W3C). *Extensible Markup Language (XML)* (Web Site). http://www.w3.org/XML

INDEX

D

D INDEX