Master Final Project Report

Done at Centre national d’études spatiales (CNES)

Model transformation with Scala

Celia Picard

18th of February 2008 – 18th of August 2008

Centre national d’études spatiales
DCT/SB/LV - BPI 2214
18, avenue Edouard Belin
31401 Toulouse cedex 9

Tutor at company
David Chemouil

Tutor at school
Silvia Clerici
Preface

This document presents the work realized for my final master project. I made this internship at the CNES. I was welcomed in the On-Board Software Team of the Ground & On Board System department by my tutor, David Chemouil. This internship lasted 6 months. My internship took place in the context of the CNES participation to the Topcased Project. The idea was to realize a framework for model transformation. From the point of view of the metamodels the models are instance of. Apart from the realization of this framework, the idea was also to evaluate the Scala language, using it to realize the application. First, I will present the context of my internship, the subject itself and the Scala language. Then I will analyze the problem and the different solutions I could bring for each part of the project. In a third part I will study in more details the chosen solution and the realized application. Finally, I will give conclusions about the Scala language and the work done.

Keywords

Modelisation, metamodellisation, object-oriented and functional programming, Scala.

Acknowledgments

Thanks to David Chemouil, my tutor at the CNES for trusting me and giving me the opportunity to realize this project. His help and advices have been precious. Thanks to the ACADIE team for the time they dedicated to help me and solve my problems. Thanks to the On-Board Software Team agents, even if I had few chances to work with them. Special thanks to Sébastien Maitrehenry, my office-mate during these six months, and who withstood me all the time. I should not forget either Guillaume Pacini who made me some very useful comment on my report. Thanks also to all the people of the DCT/TV/AV and DCT/TV/IN teams for being so good-humoured and friendly, and particularly to Mylène Ho-Tin-Noe for helping me so often even though she had no reason for that. Last but not least, I must thank a lot my tutors at the ENSIMAG and at the FIB, Marie-Laure Potet and Silvia Clérici, for their help and support during this internship and more.

Contacts

Student: Celia Picard
12 rue des filatiers
31000 Toulouse
Tel : +33 (0)5 61 32 85 14
Cel : +33 (0)6 74 45 21 34
email : celia.picard@ensimag.imag.fr

Supervisor at the CNES: David Chemouil
French Space Agency
Flight-Software Office
DCT/SB/LV - BPI 2214
18, avenue Édouard Belin
31401 Toulouse cedex 9
Tel. / Fax : +33 (0)5 612 73576 / 74559 / 74236
email : david.chemouil@cnes.fr
Contents

1 Presentation ................................................. 7
   1.1 Context of the internship .............................. 7
      1.1.1 The CNES (French Space Agency) .................. 7
      1.1.2 General idea of the Topcased project .......... 7
   1.2 Global Presentation of the Project ................. 9
      1.2.1 Introduction ..................................... 9
      1.2.2 Problem of arity, introduction of the Fiacre Language .... 9
      1.2.3 Model transformation ............................ 9
      1.2.4 Objectives of the Internship .................. 10
      1.2.5 Previous work .................................. 10
      1.2.6 Actors of this internship .................... 10
   1.3 Introduction to the Scala language ................. 11
      1.3.1 Presentation of Scala ......................... 11
      1.3.2 Learning Scala ................................. 12

2 Analysis .................................................. 13
   2.1 Global analysis of the problem ..................... 13
      2.1.1 Decomposition in different parts ............. 13
   2.2 Other objectives into the project ................ 14
      2.2.1 Integration .................................... 14
      2.2.2 Documentation ................................ 14
      2.2.3 Scala analysis ................................ 14
   2.3 The metamodels ......................................... 14
      2.3.1 Introduction to Eclipse, EMF and Ecore .......... 14
      2.3.2 Metamodel format ................................ 15
      2.3.3 Configuration file ............................. 15
      2.3.4 How to get and read the Metamodels .......... 16
   2.4 Metamodel Parsing .................................... 16
      2.4.1 Objective ..................................... 16
      2.4.2 Creation of objects: static or dynamic ? .... 17
      2.4.3 How to generate the code of the metamodel ? .. 17
   2.5 Models and model parsing ............................ 18
      2.5.1 Model format .................................. 18
      2.5.2 Objects creation: dynamic or static ? .......... 19
      2.5.3 Model parsing .................................. 19
   2.6 Model Transformation ................................. 19
      2.6.1 Requirements for model transformation ....... 19
      2.6.2 Startegy to generate the code ................. 20
   2.7 Model Pretty Printing ................................. 20
      2.7.1 Which format ? ................................ 20
      2.7.2 How to obtain it, and when ? .................. 20
3 Scaltrans

3.1 Global view of the application .............................................. 21
  3.1.1 Name of the application ............................................... 21
  3.1.2 General views of the architecture of the application .............. 21
  3.1.3 How to use the application: the workflow .......................... 23
  3.1.4 Scaltrans in numbers .................................................. 24

3.2 Study of some points of interest .......................................... 24
  3.2.1 Multiple inheritance .................................................. 24
  3.2.2 Metamodels in various files ......................................... 25
  3.2.3 Initialization in various passes ..................................... 25
  3.2.4 Generated files ...................................................... 27
  3.2.5 Model Transformation Framework ................................... 29

3.3 How to use the application? .............................................. 30
  3.3.1 Parse the metamodel .................................................. 30
  3.3.2 Parse, transform and pretty print the models ........................ 30

3.4 Application to Fiacre to Fiacre .......................................... 32
  3.4.1 Why a transformation from Fiacre to Fiacre? ....................... 32
  3.4.2 Select flattening ..................................................... 32
  3.4.3 Seq flattening and distributivity .................................. 32
  3.4.4 Transformation of If into Select ................................... 33

4 Conclusions ............................................................................. 35
  4.1 Conclusions on the Scala language ...................................... 35
    4.1.1 Scala is a young language ........................................... 35
    4.1.2 Scala is easy to learn ............................................... 35
    4.1.3 Scala is object-oriented and functional ............................ 35
    4.1.4 Multiple inheritance ................................................ 35
    4.1.5 Mild-efficiency ....................................................... 36
    4.1.6 Pattern matching .................................................... 36
    4.1.7 Global conclusions .................................................. 36
  4.2 Comparison of the result with the initial objectives of the project 36
    4.2.1 Change in the objective ............................................. 36
    4.2.2 The framework ........................................................ 36
    4.2.3 Conclusions about Scala ............................................ 36
  4.3 What remains to be done .................................................. 37
  4.4 Planning ........................................................................... 38
  4.5 Personal conclusions ....................................................... 38

Bibliography ............................................................................. 41

A Model transformation framework for the expressions example ............ 45

B Fiacre Metamodel ..................................................................... 47

C Managed and unmanaged features .......................................... 55
  C.1 What is done ................................................................. 55
    C.1.1 Steps of the application .............................................. 55
    C.1.2 Managed Scala structures .......................................... 55
    C.1.3 Managed properties of the Ecore file .............................. 57
    C.1.4 Managed properties of the helper file ............................ 58
  C.2 What remains to be done .................................................. 58
    C.2.1 Identified unmanaged Scala structures ............................ 58
    C.2.2 Identified unmanaged properties of the Ecore file .......... 59

D User’s Manual ........................................................................ 61

E Extract of the Scaladoc ................................................................ 73
Chapter 1

Presentation

1.1 Context of the internship

1.1.1 The CNES (French Space Agency)

Founded in 1961, the Centre National d’Études Spatiales (CNES) is the government agency responsible for shaping and leading France’s space policy in Europe. Its task is to invent the space systems of the future, bring space technologies to maturity and guarantee France’s independent access to Space.

CNES is a pivotal player in Europe’s space programme, and a major source of initiatives and proposals that aim to maintain France and Europe’s competitive edge. It conceives and executes space programmes with its partners in the scientific community and industry, and is closely involved in many international cooperation programmes - the key to any far-reaching space policy. Though its ability to innovate and its forward looking vision, CNES is helping to foster new technologies that will benefit society as a whole, focusing on:

- access to space
- civil applications of space
- sustainable development
- security and defense
- science and technology research

The CNES accomplishes these missions through its four centres: Head Office in Paris, the Guiana Space Centre, the Evry Space Centre and the Toulouse Space Centre. It employs more than 2500 persons amongst who 1700 in the Toulouse Space Center where my internship took place. The CNES is organized in directions and sub-directions. I was integrated in the Flight Software Office (LV) of the Ground & On-Board Systems Department (SB) of the Toulouse Space Center.

1.1.2 General idea of the Topcased project

The aim of the Topcased project (Toolkit in OPen source for Critical Applications and SystEms Development) is to

- Develop high quality open source model editors supporting an integrated development process from system specification to product architecture and implementation (including software and hardware components).
- Develop and integrate transformation and formal verification tools by improving existing techniques and theories (model checking, simulation, model transformations, etc)
• Define modeling languages according to process development phases and certification constraints

The platform developed is intended to be an open source Eclipse plugin. The project was launched by Airbus in 2004 and is now leaded together by industrial, academic and scholar partners. It is part of the Aerospace valley competitiveness cluster. The goals this project pursues are multiple:

• to minimize the development costs
• to increase competitiveness
• to speed up the putting on the market of new products
• to perpetuate methods and tools used for system and software development. One of the big problem today is that the total lifetime of an aerospace product (from beginning of development until retirement of the last device using it) is around 30 or 40 years and no software editor can commit for such a long time.
• to gather academics, industrials and service societies in order, for example, to adapt as soon as possible advances made in the academic world to industrial projects.
• to improve the ingeneers training

There are many reasons why the partners chose to develop an open source software :

• the continuity of the tools can be guaranteed (if a special feature is needed, it can be developed by anybody who would be asked to do it)
• the control on the tools can be kept
• there are no licence costs
• it is easier to introduce innovations
• the development remains in Europe (and not in Asia or in the USA)

The platform to be created is meant for various tasks (see figure 1.1):

• the creation of the model of the software. This can be done using the Topcased workshop in the AADL modelisation language.
• the transformation of the model to a (or various) model-checking system(s) language(s)
• the verification of some properties using model-checking techniques and applications
• the production of some executable code (java for example)
• the management of the project

Figure 1.1: Software Design tasks included in Topcased - source [topa]
1.2 Global Presentation of the Project

1.2.1 Introduction

One of the problems that arises in the development process introduced in the Topcased project, is the transformation from a modeling language to a model-checking language.

1.2.2 Problem of arity, introduction of the Fiacre Language

There are various (say $n$) source modeling languages (UML, AADL, and so on) and various (say $m$) target model-checking languages (depending on the property to check, one will be more efficient than another). If the transformation was done directly there would be $n \times m$ model transformations to write. The employed solution is to pass through an intermediate language. With this method, only $n + m$ transformations need to be written (see figure 1.3). The chosen language is called Fiacre. It has been developed especially for that purpose in the context of the topcased project (see [BB+08]).

![Figure 1.2: Introducing the intermediate language, Fiacre](image)

1.2.3 Model transformation

In this internship, the concrete problem to be faced was the transformation from the AADL (Architecture Analysis and Design Language) modeling language to Fiacre. But rather than focusing only on this concrete application, the spectrum of the project was enlarged to a higher level of abstraction. The idea was to create a general framework for model transformation that, given any metamodels $A$ and $B$ and a "transformation" of the structures of $A$ into structures of $B$, could transform any model instance of $A$ into a model instance of $B$ (see figure 1.3).

![Figure 1.3: General idea of the application](image)

NB: the actual transformation (from $A$ to $B$) is to be given by the user and is its entire responsibility.
1.2.4 Objectives of the Internship

During this internship, the objectives to achieve were:

- to realize a general framework for model transformation that, given the metamodels and the transformation, can transform any model of a metamodel into a model of another metamodel.

- to realize all this work using the Scala language (see chapter 1.3), try to provide conclusions about this language (argue the pros and cons basing the analysis on some points that will have to be defined) and see whether it was appropriate for this operation.

1.2.5 Previous work

In 2007, German Sancho who was doing his internship at the CNES wrote the transformation from AADL to Fiacre in the Kermeta language (only this transformation, not at a meta level). The problem is that the application done appeared to be rather inefficient at the time for real-size applications.

Moreover, his work did not pursue exactly the same objective as mine. The central part of his work was to write the transformation from AADL to Fiacre (specify how a structure transforms into another) while for me, this part is meant to be given by the user. So actually, the two internships can be seen as complementary and that is why I did not study his application more than that.

1.2.6 Actors of this internship

This internship was done together with various actors. Of course, first, with my tutor David Chemouil. However, he was more or less the only person at the CNES I worked with on this project. Apart from him, the researchers of the ACADIE team, at the Institut de Recherche en Informatique de Toulouse (IRIT - Toulouse Research Institute in Computer Sciences) of the Université Paul Sabatier (UPS) were greatly involved in the project. As they are meant to be the final users of the application (the one that will realize models transformations), it was important to work with them so that the final product would be as close as possible of their needs. In that aim, there were various meetings with them towards the internship, to present them the Scala language, the realizations and so on. I also have to emphasize the fact that they also helped me a lot on some theoretical points, as for example, the solving of the problem I had with multiple inheritance (see section 3.2.1).

I also had contacts with people of Anyware Technologies (a service society involved in the Topcased project). The stake here was double. On the one side, they are the developers of the Topcased plugin and could help me at the beginning to solve doubts regarding the Topcased tool. On the other side, I had to introduce them the Scala language and try to convince them that this language was worth looking at.

Finally, the Scala mailing lists (see 1.3.2) were a good place to ask questions regarding the Scala language, and to get answers.

Figure 1.4 sums up the interaction I had with the different actors during this internship.

Figure 1.4: Actors of the project and interaction with them
1.3 Introduction to the Scala language

1.3.1 Presentation of Scala

Scala is a multi-paradigm language, designed to integrate the object-oriented and functional paradigms. Scala was created by the Laboratoire des Méthodes de Programmation (LAMP) at Ecole Polytechnique Fédérale de Lausanne (EPFL) in 2001 and the first public version was released in January 2004. At the moment this report was written, the latest stable version is the 2.7.1 which was released on the 5th of May 2008.

Scala is object-oriented

Scala is an object-oriented language in the sense that every value is an object. It defines classes to represent the types, properties and behaviour of objects. Apart from the classical concept of class, it also defines special kinds of classes:

- abstract classes
- traits: abstract classes that cannot take constructor parameters and that can be mixed-in. They are the Scala version of the Java Interfaces, and they are meant to represent the signature of the supported methods (with default implementation or not) and variables.
- objects: classes with unique instance (Singletons). Here, the class and its instance are indistinguishable, they are the object.
- case classes: classes that are designed to be used in pattern matching. They model the usual algebraic types of functional languages.

Inheritance is allowed (classes can be subclassed) but multiple inheritance is not. Instead, Scala offers a mechanism for mixin composition, by the mean of its traits.

Scala is functional

Scala is also a functional language in the sense that every function is a value. Scala supports anonymous, higher-order, nested and curried functions.

Scala is statically typed

Scala possesses an expressive type system. It supports various features such as:

- implicit conversion (if defined)
- existential types
- abstract types
- inner classes
- bounded types (upper and/or lower bounds)
- variance annotations (co-variance, contra-variance)
- genericity: polymorphic methods and generic classes
- compound types (types can be compound directly when creating an object)

Apart from this expressive type system, Scala has a local type inference mechanism that allows to enlighten somewhat the syntax.

Scala is extensible

Scala allows to define Domain Specific Languages (DSL) and Domain Specific Embedded Languages (DSEL). It provides some constructs that make easier the definition of DSL. For example, all operators can be prefixed, infixed or postfixed.
Scala interoperates with Java

Scala is designed to be fully compatible with Java. Java code can be directly imported in Scala. The compilation of Scala code provides .class files, executable on the Java platform.

1.3.2 Learning Scala

The first thing to do during this internship, was to learn the Scala language. To do so, some documentation is available on the scala website [scala]. The scala mailing lists are also a good place where to ask questions to the very active community. The code corresponding to a metamodel of finite state automata and some functionalities attached to it has been written as a training (see listing 1.1). Beginning to program in scala was not very hard as it is very similar at the same time to Java and functional languages like Miranda or Haskell.

```scala
class Automaton {
  case class State(l: String){ override def toString() = l }
  case object InitState extends State("Initial State")
  case class Edge(from: State, to: State, label: String) {
    override def toString() = from.toString() + "→ " + label + "→ " + to.toString()
  }
  var list_edges: List[Edge] = List()
  private var list_states: List[State] = List()
  def addEdge(e: Edge) = {
    if (lookForEdge(e).isEmpty) {
      list_edges += e
      list_states += e.from
      list_states += e.to
    } else println("Warning: "+ e.toString()+ " already exists")
  }
  def lookForEdge(e: Edge) = list_edges.filter(e.equals)
  def lookForState(s: State) = list_states.filter(s.equals)
  override def toString() = list_edges.map(_.toString).mkString("\n")
}
```

Listing 1.1: Example of Scala code: extract of the example of Finite State Automaton
Chapter 2

Analysis

2.1 Global analysis of the problem

2.1.1 Decomposition in different parts

To reach the global objectives of the project, the problem has been divided into parts (they are presented here), and each one had to be analysed (see sections 2.3 to 2.7).

Metamodels

The transformations require metamodels. Decisions must be taken about their format and a way to create them. Another question will be to know if they contain all the required information and if not, find a way to gather it.

Metamodel parsing

The metamodel has to be read in order to create the corresponding Scala structures. There are several choices to make, in particular between dynamic or static (code generation) structures creation and about the method to employ to realize the parsing.

Models

Once the metamodel parsed and the corresponding Scala structures created, models are to be read. As the models are instances of a metamodel, their format will depend mainly on the choice for the metamodels format.

Model parsing

The model has to be read according to the metamodel and the corresponding objects have to be created. These objects are instances of the structures created by the metamodel parser. Here again the choice will be about dynamic or static creation and the method to use.

Model transformation

As the final objective is to transform models, a framework is needed to realize this operation. The format of this framework, of its inputs and outputs have to be discussed. This must be done interacting with the future users as this part is the one they will use the most.

Resulting model

Finally, the format of the output of the application has to be chosen.
A global view of the different steps

Figure 2.1 shows the general workflow corresponding to these steps. It is a refinement of figure 1.3. Although not known exactly, the nature of the metamodel parser output must be something that allows to parse, transform and pretty print models according to it. The content of each step will be discussed in sections 2.3 to 2.7.

![Figure 2.1: A first view of the application workflow](image)

2.2 Other objectives into the project

2.2.1 Integration

Apart from that, it would be nice, although not crucial, for the application to be integrated (or at least integratable) in the Topcased framework (for example, by a plugin).

2.2.2 Documentation

For other people to use and/or extend the application, some user and technical documentation will have to be delivered and the written code will have to be documented (scala doc).

2.2.3 Scala analysis

Finally, some conclusions about the Scala language will be given. They will lean on some points of analysis and the conclusions will concern the use of Scala for such an application.

2.3 The metamodels

2.3.1 Introduction to Eclipse, EMF and Ecore

As they are needed for this part (see section 2.3.2), let’s introduce these three tools here.

Eclipse

Eclipse is "a software platform comprised of extensible application frameworks, tools and runtimes for software development and management" (source [edl]). In fact, each functionality is implemented under a plug-in format (for example, Topcased is one).

EMF

One of those plugins is the Eclipse Modeling Framework (EMF, see [FB+04]). EMF is an environment that facilitates modeling applications and code generation. It allows to translate a diagram of, say, a metamodel into an XMI format and afterwards, to generate associated Java code. Moreover, EMF can also generate an "editor" to create models associated with the metamodel.
Ecore

Ecore is the metamodel used to describe metamodels and models in EMF. When saving the metamodel to XMI, EMF saves it respecting the Ecore metamodel.

2.3.2 Metamodel format

As this internship is part of Topcased, the metamodel format used in Topcased, Ecore, has been chosen. Metamodels under Ecore format can be generated from diagrams with EMF. This choice was also taken because the metamodels of Fiacre and AADL already existed under this format.

From now on, we will use a representation of a grammar of expressions as an example to illustrate our matter. Its metamodel is represented in figure 2.2, its Ecore translation in listing 2.1.

Figure 2.2: Metamodel of the Expressions example

<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage [...] name="expExample" nsURI="http://expExample" nsPrefix="expExample">
  <eClassifiers xsi:type="ecore:EClass" name="Exp" abstract="true"/>
  <eClassifiers xsi:type="ecore:EClass" name="Num" eSuperTypes="#//Exp">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="n" lowerBound="1" eType="ecore:EDataType [#//EInt]"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="VarDecl" eSuperTypes="#//Exp">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType [#//EString]"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="body" lowerBound="1" eType="#//Exp" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="VarRef" eSuperTypes="#//Exp">
    <eStructuralFeatures xsi:type="ecore:EReference" name="decl" lowerBound="1" eType="#//VarDecl"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Operation" abstract="true" eSuperTypes="#//Exp">
    <eStructuralFeatures xsi:type="ecore:EReference" name="arg1" lowerBound="1" eType="#//Exp" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="arg2" lowerBound="1" eType="#//Exp" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Somme" eSuperTypes="#//Operation"/>
  <eClassifiers xsi:type="ecore:EClass" name="Produit" eSuperTypes="#//Operation"/>
</ecore:EPackage>

Listing 2.1: Example of an Ecore metamodel: the Expressions example

2.3.3 Configuration file

Another problem was that the information contained in the metamodel was not complete enough to induce the exact type of Scala object to create. It would have been good to be able to specify that the structure to create was an object, a case class, and so on. To do so, there were two solutions:

- annotate directly the metamodel diagram (and so the ecore file)
- create a configuration file containing the additional information needed
The first one was the simplest and the most concise, as no additional file nor additional treatment was required (apart from finding the information in the metamodel). However, it meant that the metamodel itself would contain information directly related to its Scala implementation. This did not seem correct as the metamodels would serve for other things. Beside, this would have meant changing the existing metamodels. So, instead it was chosen to create a configuration file. Indeed, even if it implied having an additional file and additional treatment it allowed to separate the metamodel from its implementation. The metamodel of the chosen format is described in figure 2.3.

![Figure 2.3: Scala Helper metamodel](image)

Listing 2.2 shows a concrete example of configuration file. All the classes are reproduced with their names and some additional attributes indicating whether they are case classes, objects, and so on.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Package name="etestCelia" ref="/EEPackage">
  <Class name="Exp" ref="/EEPackage/eClassifiers.0" isCaseClass="false" isObject="false"/>
  <Class name="Num" ref="/EEPackage/eClassifiers.1" isCaseClass="true" isObject="false"/>
    <Attribute xsi:type="ecore:EReference" name="body" ref="/EEPackage/eClassifiers.1"/>
  </Class>
  <Class name="VarDecl" ref="/EEPackage/eClassifiers.2" isCaseClass="true" isObject="false"/>
    <Attribute xsi:type="ecore:EReference" name="decl" ref="/EEPackage/eClassifiers.0"/>
  </Class>
  <Class name="VarRef" ref="/EEPackage/eClassifiers.3" isCaseClass="true" isObject="false"/>
    <Attribute xsi:type="ecore:EReference" name="body" ref="/EEPackage/eClassifiers.1"/>
  </Class>
  <Class name="Operation" ref="/EEPackage/eClassifiers.4" isCaseClass="false" isObject="false"/>
    <Attribute xsi:type="ecore:EReference" name="arg1" ref="/EEPackage/eClassifiers.0"/>
    <Attribute xsi:type="ecore:EReference" name="arg2" ref="/EEPackage/eClassifiers.1"/>
  </Class>
  <Class name="Somme" ref="/EEPackage/eClassifiers.5" isCaseClass="true" isObject="false"/>
  <Class name="Produit" ref="/EEPackage/eClassifiers.6" isCaseClass="true" isObject="false"/>
</Package>
```

Listing 2.2: Example of a configuration file for the Expressions example

### 2.3.4 How to get and read the Metamodels

The metamodels are stored in files (under Ecore format). These files can be read by the functions of the predefined package `scala.xml.parsing`. If needed, the metamodels can be generated thanks to EMF by drawing the metamodel diagram. EMF generates the associated Ecore file.

### 2.4 Metamodel Parsing

#### 2.4.1 Objective

The objective to achieve in this step is to create the Scala structures corresponding to the structures of the Ecore metamodel (see 2.3). To do so, the metamodel has to be parsed (read it and analyse its content). The reason why Scala structures have to be created is the following. We must not forget that the main objective the application we are creating pursues is to do model transformation,
and we want to manipulate Scala objects during model transformation. But the models we are manipulating are instances of metamodels. So the first thing, to be able to create those instances, is to have the metamodels in Scala.

2.4.2 Creation of objects : static or dynamic ?

The very first choice to make was to decide whether the scala classes corresponding to the metamodel were to be created dynamically or statically. Dynamically would have meant that they were created while parsing and that once the parsing was over, the objects would exist in the system and would be usable directly. That was a positive point because it meant that no additional compilation was needed. However, that also meant that these objects had to be created each time a model transformation was done. That meant make the parsing each time, while the idea was that the same metamodels would be used quite often and that a lot of models would be transformed, at different moments, without needing to parse the metamodels each time. For that reason, static code generation was chosen. It has to be compiled to parse and transform models. This code is the Scala code corresponding to the Ecore metamodel. What’s more, this decision was consistent with the code generation policy of EMF, as we will see in the next section (see 2.4.3).

However, me must make an important remark here. Static code is not generated directly from the metamodel. Actually, what seemed the better solution was to implement, in Scala, a representation of the Scala metamodel and to create during the metamodel parsing instances of this metamodel. Then in the end, pretty print these instances in Scala code.

We will see later that actually more code than just that will need to be generated during parsing.

2.4.3 How to generate the code of the metamodel ?

The different possible solutions

Once taken the decision to generate static code, the question of how to do it rose up. Once again, there were various solutions, partly because EMF provides features that allow to generate Java code directly from an Ecore metamodel. The different options were :

Solution 1 use directly the java code generated by EMF (possible as Java and Scala are fully compatible)

Solution 2 wrap the code generated by EMF in Scala structures

Solution 3 adapt the EMF libraries so that it generates Scala code instead of Java.

Solution 4 write a program that generates Scala code directly (write the program from scratch)

Analysis of each solution

Solution 1 This was obviously the simplest solution but it presented very low interest. Indeed, the idea was to use Scala in order to make the most of its properties so using Java code would make this completely impossible. This was definitely not the good solution here.

Solution 2 This solution had various advantages. It offered the possibility to reuse the EMF code generator and the only thing to do was to embed the generated code in Scala code. Moreover, only a part of the code could have been embedded with specific Scala code and the rest could have remained as is and used directly. However, this solution would have been quite heavy precisely because of this embedded code. What’s more, this also meant understanding the whole code generated by EMF. What must be noted here is that this code is very complete, there are a lot of tools, factories, and so on and there are still risks of misunderstanding. Finally, as said before the structures generated by EMF are very complete and powerful but way too much for the needs here. Efficiency might be gained if not reusing them. Another thing to add is that EMF generates a lot of structures that are needed when using Java but that can be programmed in a lighter way when using Scala (for example visitor pattern in Java can be implemented using pattern matching in Scala and the result is much simpler).
Solution 3  Another solution, instead of adapting the generated code, would have been to adapt the code generation. That is, modify the EMF libraries that generate the code (or extend them) so that instead of generating Java code it would generate Scala code. This would have solved the problem of heaviness but not the problem of complexity. Moreover, the code generator or the generated code could still be misunderstood. Beside, one of the objectives of the project was to test the Scala language. In particular to test it for model transformation. Here, the code generator if adapting the EMF libraries, would have been written in Java (even if it generated Scala code) which did not complete entirely with the given objectives.

Solution 4  This solution was the one that reused the least existing solutions (all the other ones reused EMF, this one not). But it allowed to write the program in Scala and to create the exact Scala structures needed.

The chosen solution
As the project did not require a lot of complex structures and that the idea was to take as much advantage as possible of the Scala language, the last solution was chosen. Of course, it has disadvantages, mainly because the code generation has to be re-thought from scratch. But only useful code is generated and it is done to be as efficient as possible. Another aspect for which this solution was chosen was that there is a pre-existent library in Scala that allows to parse XML code to Scala (scala.xml.) and store it in a tree structure. So what has been called the metamodel parser until here will in fact consist in analyzing the content of the above mentioned tree. The Scala code resulting of the parsing of the metamodel of the expressions example (see figure 2.2 and listing 2.1), is shown in listing 2.3. Each class is created, with its super class (if existant) and attributes.

<table>
<thead>
<tr>
<th>package expExample</th>
</tr>
</thead>
<tbody>
<tr>
<td>trait Exp</td>
</tr>
<tr>
<td>case class Num(n: scala.Int) extends Exp</td>
</tr>
<tr>
<td>case class VarDecl(name: Predef.String, body: Exp) extends Exp</td>
</tr>
<tr>
<td>case class VarRef(dec: VarDecl) extends Exp</td>
</tr>
<tr>
<td>abstract class Operation(val arg1: Exp, val arg2: Exp) extends Exp</td>
</tr>
<tr>
<td>case class Somme(override val arg1: Exp, override val arg2: Exp) extends Operation(arg1, arg2)</td>
</tr>
<tr>
<td>case class Produit(override val arg1: Exp, override val arg2: Exp) extends Operation(arg1, arg2)</td>
</tr>
</tbody>
</table>

Listing 2.3: Generated Scala code for the Expressions example

2.5 Models and model parsing
Once determined the metamodels format and how they are parsed, the same things must be decided for models. However, the choices may highly be linked to the choices made for metamodels.

2.5.1 Model format
The model is logically an instance of the input metamodel. As Ecore was chosen to be the metamodel input format, the model input format has to be an instance of the metamodel, written with an XMI syntax. To generate models according to a metamodel, EMF can be used as it offers to generate an editor for a given metamodel (Topcased even offers to generate a graphical editor. But as this is secondary, it will not be detailed here. However, there are some tutorials that show how to achieve it see [topb].).
CHAPTER 2. ANALYSIS

Listing 2.4 shows an example of a model for the expressions example. It corresponds to the expression 
\( (\text{val } b = 4) + b + 0 \)

|expExample:Somme xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmns:expExample="http://expExample">
   | <arg1 xsi:type="expExample:Somme">
   |   | <arg1 xsi:type="expExample:VarDecl" name="b">
   |   |   | <body xsi:type="expExample:Num" n="4"/>
   |   | </arg1>
   |   | <arg2 xsi:type="expExample:VarRef" decl="/@arg1/@arg1="/>
   | </arg1>
   | <arg2 xsi:type="expExample:Num"/>
</expExample:Somme>

Listing 2.4: Model example for the Expressions example in XMI format

2.5.2 Objects creation: dynamic or static?
Here again, it had to be decided if the structures had to be generated statically or dynamically. Unlike metamodels, same model will not be used various times. The parsed model is to be transformed, and pretty printed directly and that chain is to be executed without intervention. So clearly here, instances had to be created dynamically. No static code has to be generated during parsing. The object resulting of the parsing is to be passed as is to the input of the transformer.

2.5.3 Model parsing
The same way functions of the package scala.xml were used for metamodels parsing, they were used for models parsing. However, in the metamodel case, the information to find in it was known (the one corresponding to the Ecore metamodel) whereas here, the information corresponded to the metamodel of which the model is an instance. In order to read efficiently the model and to create the appropriate structures, additional information from the metamodel was needed (for example, the name of the attributes). The easiest way was to get it when parsing the metamodel.

The output of the model parser is the Scala object corresponding to the input XMI model. You can see in listing 2.5 the result of parsing the model of listing 2.4.

```
Somme(Somme(VarDecl(b, Num(4)), VarRef(VarDecl(b, Num(4)))), Num(0))
```

Listing 2.5: Result of the parsing of the model 2.4 of the Expressions example

2.6 Model Transformation
Even if not necessarily the most complicated to realize, this was the heart of the application. We must not forget that all the project aims at transforming a model instance of a metamodel A into a model instance of a metamodel B.

We must insist here again on the fact that the transformation is to be given by the user. The only thing this application offers is a way to express this transformation and apply it. However, the way a structure transforms into another has to be written by the person who wants to realize the transformation.

2.6.1 Requirements for model transformation
As the transformation is written by the user, a transformation framework was to be made that will be enhanced afterwards by the user. So what was very important here was to be able to capt the needs and desires of the users for the framework to be useful for them.

The users are the researchers of the ACADIE team of the Institut de Recherche en Informatique de Toulouse (IRIT) of the Université Paul Sabatier (UPS) in Toulouse. We had various meetings with them to try to understand what the transformation framework had to look like. This has been a bit complicated as they had a pretty precise idea of what they wanted as they had worked with German Sancho (see 1.2.5) last year and helped him with his model transformation system. They wanted
something quite alike but the languages used and the levels of abstraction were very different. There were many problems to be faced in order to achieve something similar to what they expected. However, here is a brief idea of what is needed to make model transformation:

- an input model, as a Scala object conforming to the input metamodel
- a signature for the transformation that is inherited when defining a concrete transformation
- an iterator that iterates the transformation on all parts of the model
- an output model, as a Scala object conforming to the target metamodel. It is important for the output to be under the same format as the input because this would allow to realize with no intermediate operation various transformations one after the other.

2.6.2 Strategy to generate the code

Here, it was obvious, as one might want to make various transformations and as the user has to make them himself, that static code is needed. But the question that remained was when to generate this code? The easiest probably was to generate it when parsing the metamodel because that is when all the information about the different classes is available. So, this was another operation to be realized by the metamodel parser.

2.7 Model Pretty Printing

This is the last part of the whole model transformation process. Once the model transformed, remained the question of under which format it had to be returned (in order to be able to re-use it for example) and how to make this last "transformation".

2.7.1 Which format?

One interesting thing, as said already, would be that the output of a transformation process could be re-injected as the input of another one. This would make things consistent and easier to use. Therefore, the output format chosen is the same as the input format (for the whole application) which is an XMI format that conforms to the target metamodel (metamodel target of the transformation). As an example, the output of the application for a transformation that only clones the model (that is, that makes no transformation actually) would be equal to the input model.

2.7.2 How to obtain it, and when?

The same way some information taken directly in the metamodel was needed to realize the model parsing, some additional information is necessary to realize the model pretty printing. We will see later what exactly is needed, but once more the easiest way to get this information was to get it while parsing the metamodel. So this was again another task to realize during the metamodel parsing.

Moreover, the same way the package _scala.xml_ contained features to read XMI from a file and transform it to Scala objects (nodes), it contains features to transform Scala objects (_scala.xml.nodes_) into XMI and write it into a file. Of course, these functionalities were used and the work here was to transform the Scala objects (that are instances of the target metamodel) into instances of _Nodes_.

Chapter 3

Study of the chosen solution: the *scaltrans* application

3.1 Global view of the application

3.1.1 Name of the application

As the application is to be integrated in the Topcased project, it needed a name. *Scaltrans* was chosen.

3.1.2 General views of the architecture of the application

We saw (chapter 2) the different steps to implement in order to realize the whole model transformation process.

View of a first application

![Diagram of the architecture of the first application]

Figure 3.1: Architecture of the first application
The first application that was made, only implemented these different steps. However, it had been noticed that for model parsing a certain number of operations were common to all metamodels and others required information specific to each metamodel. So a model parser that implemented the generic features and that was inherited by the model parser generated by the metamodel parser was created. The same thing applies for the pretty printing operation.

For the transformation framework, until the very end it was not clear what exactly it had to look like (the input and output were known, together with what it had to do but not the details of how it had to be implemented).

Figure 3.1 represents the architecture of this first application (first step of the iteration process applied during this internship). This application has been enhanced afterwards, as we will see further.

In this first version, we can already see the separation between the two parts of the application: the metamodel parsing and the model parsing. We also see the way the model parser and pretty printer are generated by the metamodel parser. The parts called MMScalaHelper and MMParserHelper are the parts that manage the configuration file (also called sometimes helper file).

The files called GXX and GYY represent the Scala implementation of the metamodels XX and YY.

**View of the final application**

After this first iteration, it was clear that things could be abstracted a little bit more than what had been done. Indeed, all the operations (parsing, pretty printing) can be seen as transformations. Careful here, we are not talking about the transformation that is the heart of the application. Let’s explain this idea. If we move to a higher level of abstraction than before, the parsers (for metamodels and models) can be seen as boxes that take an XMI model as input and that return the corresponding model as Scala objects (we must not forget that actually the output of the metamodel parser is an instance of the metamodel Scala that is pretty printed as Scala code afterwards). In short, this operation corresponds to a transformation from Nodes to instances of a metamodel (respectively, the Scala metamodel or the input metamodel).

The same way, the model pretty printing can be seen as the transformation from instances of a metamodel to instances of Nodes.
Once this was clear, the treatment of these operations was normalized. Actually, the implementation was quite particular to each case but an interface (T2U) that could be extended by all classes realizing these operations was created.

In the end, the final architecture obtained is the following (figure 3.2). We can add various commentaries about this new architecture.

1. It is important to insist on the fact that even if they are called "transformations", these operations are not related with the models transformation framework. Actually, this class does not implement the interface T2U. There are various reasons for that. First, the format of T2U did not fit the format the users asked me for the transformation framework. There is also a crucial difference between these operations. For parsing and pretty printing, the transformation is known and written by me, while in the other case the transformation is written by the user. Anyway, this does not mean that the framework for model transformation could not enter in the frame of the interface T2U but I did not have enough time to think about it really.

2. Another important change had to be made in the architecture because the metamodels can be stored in various files with crossed references. The application had to be embedded into a wrapping application so that it would be applied correctly to all files of the metamodel. This is what the class WholeMM2Scala is for.

3. Another level of abstraction, differentiating between the transformations from Nodes to metamodels instances, and the transformations from these instances to Nodes was also added.

4. Various classes had their names changed: MMParser became Ecore2Scala, MMParserHelper became Ecore2ScalaHelper, Parser became Model2Scala.

3.1.3 How to use the application: the workflow

Now that we have seen the global form of the application, it can be interesting to see exactly how to use it. The following diagram (figure 3.3) gives the workflow to follow in order to realize all the different steps implemented during this internship. However, it is important to remind that the operations concerning the metamodels and the models can be realized separately (as long as when treating a model, the corresponding metamodels have been parsed before).

![Figure 3.3: Workflow of activities to execute in order to realize a whole model transformation](image-url)
3.1.4 Scaltrans in numbers

The organization in packages and classes of the application is shown in figure 3.4.

![Organization diagram](image)

Figure 3.4: Organization in packages and classes of Scaltrans

As we can see, the application is made of 4 packages and the generated files make a fifth one. In total, there are 13 static classes and as we already know, the metamodel parser generates four more classes. In total (commentaries included), the 13 static classes represent around 4400 lines of code. The smallest class \((XMI2Scala)\) only has 34 lines of code while the largest one \((MMScala)\) has 1324. Last thing that should be mentioned here is that all this code was written in Scala.

My internship consisted of the realization of all the work mentioned in this document.

3.2 Study of some points of interest

Although we won’t study in details all the work done, some points deserve to be discussed here.

3.2.1 Multiple inheritance

![Multiple inheritance diagram](image)

Figure 3.5: Example of management of multiple inheritance

24
As said in section 1.3, multiple inheritance is not allowed in Scala. Instead, Scala offers a mix-in mechanism. A class may only have one super class but it can mix-in various traits\(^1\). But in the studied metamodels, there was multiple inheritance. Of course, the first solution would have been to translate all abstract classes with no attributes to traits (and not only Interfaces). This was enough for the Fiacre metamodel, but not for AADL. To solve this problem, intermediate traits were created as shown in figure 3.5. We can note the propagation phenomenon. Indeed traits must be created for all super classes of classes involved in a context of multiple inheritance.

### 3.2.2 Metamodels in various files

![Figure 3.6: Management of the metamodels in various files](image)

In section 3.1.2 we saw that the metamodels could be split in various files, one per package. A class may reference a class contained in another package. So, the classes need to have access to classes of other packages. Figure 3.6 explains the implemented solution. In outline, each package has a table that references all classes of the package. The cases of all these tables are themselves referenced in one global table. Thanks to this table, a class can have access to all other classes.

### 3.2.3 Initialization in various passes

A problem that rose up was that one might need to pass various times over the input to initialize the structures he is creating (this is true for metamodel parsing, as well as for model parsing or pretty printing). For example, if in a metamodel there is a reference to a class over which the tree parser has not come across yet, the reference will have to be initialized later. To do so, an equivalent of the compilators symbol tables was created. When coming across a new feature, all the information about it is stored in a table (together with an indicator saying whether all this information could be initialized). Once the whole tree has been read, the program comes back on the nodes that could not be fully initialized and try to initialize them thanks to the information stored until here. This is repeated until all structures are initialized. When the metamodel is stored in various files, all the files must be treated once before the second pass. The structures can only be created at the very end, when all information has been gathered. Class symbTab of T2U defines all the elements needed for that operation (see listing 3.1). The elements needed for initialization will be introduced in a class extending cSymTab.

\(^1\)Recall: traits are like Java interfaces, they are abstract classes with no constructor attributes
Listing 3.1: The class `symbTab` of trait `T2U`

Figure 3.7 shows the algorithm to follow. The only thing not mentioned until now is the `setPropForAll` function. It appeared that there might be some properties that may only be set once everything has been initialized and before the phase of object creation. For example, the indication of when creating a trait. If there are no such properties, `setPropForAll` does not do anything.
3.2.4 Generated files

As said in chapter 2, various files are generated during metamodel parsing. We will not discuss here any further the one that contains the Scala code representing the metamodel, as it has already been explained in section 2.4. We will concentrate on the files generated for model parsing and pretty printing. The model transformation framework, as for it, will be explained in section 3.2.5.

Generated file for model parsing

As explained in section 2.5, additional information is needed from the metamodel to parse the model. When parsing the model, the manipulated objects are strings even if under the form of a tree. The needed information will permit to create the Scala objects associated with these strings.

In concrete terms, functions will link, by the mean of pattern matching, the class name with the needed information. This information is of two kinds: information about the attributes of each class (name, type, kind (reference, data, containment), default value, arity) on the one hand and, on the other hand, access to functions that create the Scala objects corresponding to what is being parsed. Finally, after parsing there are some checkings to do on the correctness of the parsed model with regard to its metamodel. Of course, some properties may be checked statically during object creation (for instance, existence of a non null attribute) but others cannot (for example, arity of a multiple attribute). Naturally, for models created with the Topcased editors, these checkings are unnecessary, but as models may be created by other means, these verifications are always made.

The generated file extends the model parsing class, Model2Scala (see 3.2), and implements the functions whose signature is defined in this framework (and additional functions if needed). As an example, listing 3.2 shows the code generated for a part of the expressions example.

```scala
package expExample
import org.topcased.scaltrans.parser.Model2Scala
class EXPEXAMPLEParser extends Model2Scala[Any]{
  import scala.runtime.RichString
  import scala.xml._

  override val packageName: Predef.String = "expExample"

  def functionToUse(name: Predef.String): ((X, X, X) => Any) = name match {
    case "Somme" => createSomme _
    case "VarDecl" => createVarDecl _
    case _ => error("...")
  }

  def listAttr(name: Predef.String): List[tAttr] = name match {
    case "Somme" => List(new tAttr("arg1", "Exp", "OC", null, "Unique"),
                        new tAttr("arg2", "Exp", "OC", null, "Unique"))
    case "VarDecl" => List(new tAttr("name", "", "simple", null, "Unique"),
                        new tAttr("body", "Exp", "OC", null, "Unique"))
    case _ => error("...")
  }

  def createSomme(lattrS: X, lattrC: X, lattrR: X):Somme = lattrC match {
    case List(List(arg1), List(arg2)) =>
      Somme(if (arg1 != null) arg1.asInstanceOf[Exp] else null,
             if (arg2 != null) arg2.asInstanceOf[Exp] else null)
    case _ => error("...")
  }

  def createVarDecl(lattrS: X, lattrC: X, lattrR: X):VarDecl = lattrS match {
    case List(List(name)) => lattrC match {
      case List(List(body)) =>
        VarDecl(name.toString, if (body != null) body.asInstanceOf[Exp] else null)
      case _ => error("...")
    }
    case _ => error("...")
  }

  def verify (e: Any): Boolean = e match {
    case Somme(a, b) => verifySomme(e.asInstanceOf[Somme]) && verify(a) && verify(b)
    case VarDecl(n, b) => verifyVarDecl(e.asInstanceOf[VarDecl]) && verify(b)
  }
}
``
```scala
case _ => error("...")
}

def verifySomme(exp: Somme): Boolean = {
  if (exp.arg1 == null) error("...")
  if (exp.arg2 == null) error("...")
  true
}

def verifyVarDecl(exp: VarDecl): Boolean = {
  if (exp.body == null) error("...")
  true
}
```

Listing 3.2: Generated file for model parsing for a reduced Expressions example

**Generated file for model pretty printing**

The same way additional information is needed for model parsing, additional information is needed for model pretty printing. Here, the operation is the opposite to the parsing one. Scala objects are to be transformed into strings. In concrete terms, the needed information is: the names, types and kind (reference, data, containment) of the attributes and the classes names. As an example, listing 3.3 shows the code generated for the expressions example.

```scala
class EXPEXAMPLEPrettyPrint extends PrettyPrint[Any] {
  import scala.runtime.RichString
  import scala.xml_

  val prefix = "expExample"
  val xmlns = "http://expExample"

  def listAttrOC(e: Any): List[(String, Any, Int, String, String)] = e match {
    case expExample.Somme(arg1, arg2) =>
      Scala2Node.createListAttrOC(arg1, "arg1", "null", "Unique") ::
      Scala2Node.createListAttrOC(arg2, "arg2", "null", "Unique")
    case expExample.VarRef(dec) => List()
    case expExample.VarDecl(name, body) =>
      Scala2Node.createListAttrOC(body, "body", "null", "Unique")
    case _ => error("...")
  }

  def listAttrRef(e: Any): List[(String, String, Boolean)] = e match {
    case expExample.Somme(arg1, arg2) => List()
    case expExample.VarRef(dec) => Scala2Node.lookUpRef(dec, "decl")
    case expExample.VarDecl(name, body) => List()
    case _ => error("...")
  }

  def getNameClass(e: Any): Predef.String = e match {
    case expExample.Somme(arg1, arg2) => "Somme"
    case expExample.VarRef(dec) => "VarRef"
    case expExample.VarDecl(name, body) => "VarDecl"
    case _ => error("...")
  }
}
```

Listing 3.3: Generated pretty printing file for an Exp. example with Somme, VarRef and VarDecl

28
3.2.5 Model Transformation Framework

Finally, the last point we will tackle in this section is the framework for model transformation. It is the heart of the application, even if not the part most time was dedicated to.

This is the crucial part for the users and that is why the outline of this framework had to be defined with them. Here is a serie of some problems that were faced and the solution brought.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are the input and output types of the transformation functions?</td>
<td>Abstract types were used, with relations between them for the global inheritance tree to be preserved. These types are materialized when creating a transformation</td>
</tr>
<tr>
<td>How is the iteration of the transformation over an object attributes done (manually by the user when he writes a transformation or automatically)?</td>
<td>An iterator was made apart from the transformation function itself. This iterator role is to iterate the transformation over the attributes and to call the transformation function of the object with the transformed attributes</td>
</tr>
<tr>
<td>Is a global transformation function needed (with pattern matching, calling the appropriate function) or not? It would always be called by the user when initiating a transformation.</td>
<td>The solution where the user calls the iterator corresponding to the object he is transforming was chosen. Even though it would have been good to call always the same function, no matter the type of the input object, the typage obtained thanks to the abstract types would have been lost.</td>
</tr>
<tr>
<td>A point outlined early was related with the reference attributes. When finding one, the thing to do is to get its transformed form, and not to create it again. So the problem was how to memorize these objects, when, where, and who has to do it.</td>
<td>A hash map associating the non transformed object with its image was used. When finding a reference attribute, the program goes and finds the appropriate object in the hashmap. Actually, in concrete terms one hashmap is created per type that can be referenced (subtypes included) and makes the memorization only for those types. These operations are done automatically by the iterator.</td>
</tr>
</tbody>
</table>

The objects contained in the model transformation framework are the following:

- a trait `Signature` that defines the abstract types representing the output types and the signature of the `transformXX` function for each class of the metamodel. When creating a new transformation, this class must be extended and its features redefined.

- a class `Iterator` that has a `Signature` attribute and that defines the hashmaps and the iterator function, `iterXX`, for each class that needs one. This iterator function calls the appropriate iterator function over each attribute and `transformXX` with the transformed attributes.

- a class `Identity` extending `Signature` and defining the Identity transformation.

- an object `Clone` that extends `Iterator` with attribute `Identity`. One will need to call the methods of this object to apply the `Identity` transformation as `Clone` contains the iterators. However, `Clone` does not redefine any of the functions of `Iterator`.

As a concrete example, you will find in appendix A the generated framework for the expressions example.

---

2the classes that will need an iterator are the one that can be instanciated as an attribute of another class or that can be the class of the object beind parsed.
3.3 How to use the application?

Now, we will see a concrete example of usage of the application, still on the Expressions example.

3.3.1 Parse the metamodel

The first thing to do is to parse the metamodel. To do so, function `WholeMM2Scala` must be called with the appropriate attributes:

- the name of the global package (serves only if the metamodel has various files, otherwise can be "")
- a list of tuples containing, for each file of the metamodel, the path to the configuration file, the path to the file of the metamodel and the file where to print the metamodel Scala code corresponding to this metamodel.
- a list of tuples containing, for each file of the metamodel, the path to the files where to print the model parser, the model pretty printer and the model transformation framework.

Listing 3.4 shows an example of code realizing these steps.

```scala
object MMExpParsing {
  import org.topcased.scaltrans.mmparser.WholeMM2Scala
  def main(args: Array[String]): Unit = {
    var args = List(("configExp.ecore", "mmexp.ecore", "Exp.scala"))
    var files = List(("ExpParser.scala", "ExpPP.scala", "ExpTransfo.scala"))
    WholeMM2Scala.parser("", args, files)
  }
}
```

Listing 3.4: How to parse a metamodel

After this, the code of the metamodel in Scala, for model parsing, model pretty printing and model transformation are located respectively in `Exp.scala`, `ExpParser.scala`, `ExpPP.scala` and `ExpTransfo.scala`. These files have to be compiled to be usable.

3.3.2 Parse, transform and pretty print the models

Listing 3.6 shows how to make model parsing and transformation. For better comprehension, we will explain here the transformations made (see listing 3.5). Two of them are defined:

- the first one, `InverseSomme`, only reverses the terms of a `Somme`. As the target types are not changed in regard to `Identity`, the function `transformSomme` can be redefined directly.

- the second one, aims at transforming a `Somme` into `Produit`. So the target types must be changed and for that reason, the transformation can’t inherit from `Identity` but from `Signature`. Here, the types are invariant for inheritance. That is why all the types and `transformXX` functions have to be redefined even if only one type and one function change in regard to `Identity`.

As said in section 3.2.5, a class that inherits from `Iterator` has to be defined in order to create a transformation.

```scala
object InverseSomme extends Identity {
  override def transformSomme(a1: TargetExp, a2: TargetExp): TargetSomme = Somme(a2, a1)
}

object TransfoProduit extends Signature {
  type TargetProduit = Produit
  type TargetSomme = Produit
```
CHAPTER 3. SCALTRANS

```scala
type TargetOperation = Operation
type TargetVarRef = VarRef
type TargetVarDecl = VarDecl
type TargetNum = Num
type TargetExp = Exp

override def transformProduit(arg1: TargetExp, arg2: TargetExp): TargetProduit = Produit(arg1, arg2)

override def transformSomme(a1: TargetExp, a2: TargetExp): TargetSomme = Produit(a2, a1)

override def transformVarRef(dec: TargetVarDecl): TargetVarRef = VarRef(dec)

override def transformVarDecl(n: String, b: TargetExp): TargetVarDecl = VarDecl(n, b)

override def transformNum(n: scala.Int): TargetNum = Num(n)
```

```scala
object TransfoExp1 extends Iterator(InverseSomme)
object TransfoExp2 extends Iterator(TransfoProduit)
```

**Listing 3.5:** Example of concrete model transformations

Listing 3.6 shows how to use these transformations. First the model is parsed, then it is transformed with both transformations and finally the results are pretty printed.

```scala
object ModelExpParsing {
  import eTestCelia._
  def main(args: Array[String]): Unit = {
    var files = List("ExpExample/expTest.eTestCelia")
    val eParser = new ETESTCELIAParser
    val ep = eParser.parserFromFile(files)
    val cloneep = Clone.iterExp(ep.asInstanceOf[Exp])
    var tep = TransfoExp1.iterExp(ep.asInstanceOf[Exp])
    while (TransfoExp1.otherPass) {
      TransfoExp1.otherPass_=(false)
      tep = TransfoExp1.iterExp(ep.asInstanceOf[Exp])
    }
    var ttep = TransfoExp2.iterExp(tep.asInstanceOf[Exp])
    while (TransfoExp2.otherPass) {
      TransfoExp2.otherPass_=(false)
      ttep = TransfoExp2.iterExp(tep.asInstanceOf[Exp])
    }
    val ePrettyPrintBefore = new ETESTCELIAPrettyPrint
    val ePrettyPrintTransfo1 = new ETESTCELIAPrettyPrint
    val ePrettyPrintTransfo2 = new ETESTCELIAPrettyPrint
    val ppep = ePrettyPrintBefore.prettyPrint(ep)
    val pptep = ePrettyPrintTransfo1.prettyPrint(tep)
    val ptttep = ePrettyPrintTransfo2.prettyPrint(tttep)
  }
}
```

**Listing 3.6:** Example of model parsing and transformation

We can note that the transformation is iterated over. This is done, as for parsing and pretty printing, to initialize all the structures (in particular all the reference attributes) even if the structures they refer to are created after them. The only thing is that here this iteration must be written explicitly by the user, because the program does not know over which function to iterate.

**Listing 3.7** shows the pretty printed result of the second transformation. The input model is the one shown in listing 2.4.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<expExample:Produit xmlns:expExamples="http://expExample">
  <arg1 xsi:type="expExample:Num" n="0"></arg1>
  <arg2 xsi:type="expExample:Produit">
    <arg1 xsi:type="expExample:VarRef" decl="/@arg2/@arg2"></arg1>
    <arg2 xsi:type="expExample:VarDecl" name="b">
      <body xsi:type="expExample:Num" n="4"></body>
    </arg2>
  </arg2>
</expExample:Produit>
```

**Listing 3.7:** Result of the second transformation
3.4 Application to a concrete case: Fiacre to Fiacre transformation

3.4.1 Why a transformation from Fiacre to Fiacre?

In section 1.2.3, it is said that the concrete application the project aims at was the transformation from model written in AADL to models written in Fiacre. This was a guide during the whole internship and both metamodels were used and studied so that this transformation could eventually be made.

However, it probably was too ambitious an objective as the AADL metamodel is large and has many specificities. All of them could not be managed, only the most important ones. For example, the peculiarity that the metamodel AADL is divided in various files as explained in section 3.2.2. The application has been prepared so that it would not be too complicated to introduce the new features, but I did not have time to fully understand them and chose not to enter in it rather than to make something false. In the annex C you will find the list of features supported by the application and the list of the non supported ones that could be identified.

The fact that the whole AADL metamodel could not be managed is due to the iterative way of proceeding used in this internship. New features were implemented when they were found in a metamodel. This has had the advantage that the system was enlarged step by step, so in a more secure way.

Anyway, the people of ACADIE were interested too by transformations from Fiacre to Fiacre, to simplify some features. Some of these transformations will be explained now.

3.4.2 Select flattening

Fiacre’s Select is a way for non deterministic choice. When there are various nested Selects, the idea is to flatten them into one Select (see fig. 3.8). Listing 3.8 shows the associated Scala code.

```
override def transformSelect(st: List[TargetStatement]): TargetSelect = {
  def f(s: List[TargetStatement]): List[TargetStatement] = s match {
    case Select(x) :: l => x :: f(l)
    case x :: l => x :: f(l)
    case List() => List()
  }
  Select(f(st))
}
```

Figure 3.8: Illustration of Select flattening

3.4.3 Seq flattening and distributivity

The Fiacre Seq represents a sequence of instructions that have to be executed the one after the other. Apart from flattening various nested Seq (similar to 3.4.2), when in a Seq there is one
instruction and then one select, the idea is to distribute the first instruction over the select (see fig. 3.9). Listing 3.9 shows the corresponding Scala code.

Figure 3.9: Illustration of Seq distributivity

```scala
override def transformSeq(statements: scala.List[TargetStatement]): TargetSeq = {
  def transfoSelectInSeq(s: List[TargetStatement]): List[TargetStatement] = s match {
    case Seq(a) :: otherSta => a :: transfoSelectInSeq(otherSta)
    case a :: Select(s) :: os => {
      def f(b: TargetStatement) = b match {
        case Seq(c) => Seq(a :: c)
        case _ => Seq(List(a, b))
      }
      Select(s.map(f)) :: transfoSelectInSeq(os)
    }
    case s :: otherSta => s :: transfoSelectInSeq(otherSta)
    case _ => List()
  }
  if (statements.length == 1) statements.head
  else {
    var res = transfoSelectInSeq(statements);
    var old = statements
    while (res != old) {
      old = res;
      res = transfoSelectInSeq(old)
    }
    Seq(res)
  }
}
```

Listing 3.9: Scala implementation for Seq distributivity

### 3.4.4 Transformation of If into Select

The last example we will present here is the systematic transformation of If into Select. The reason for this is that when an If is nested into a Select, if it is transformed into a Select, then the transformation for nested selects can be applied. Figure 3.10 and listing 3.10 explain this transformation.

Figure 3.10: Illustration of the transformation of If into Select

```scala
override def transformIfStmt(c: TargetExp, t: TargetStatement, e: TargetStatement): TargetIfStmt = Select(List(Seq(List(DeterministicAssignment(List(SingleAssignment(BoolLiteral(true), c.asInstanceOf[Exp]))), t))), Seq(List(DeterministicAssignment(List(SingleAssignment(BoolLiteral(false), c.asInstanceOf[Exp]))), e)))
```

Listing 3.10: Scala implementation for transformation of If into Select
Chapter 4

Conclusions

To close this report, we will see the conclusions that could be drawn from our experience, we will compare the result and the actual planning of this internship with the objectives and planning that were set at the beginning of this internship.

4.1 Conclusions on the Scala language

To complete with the objective of evaluating Scala, here are some evaluation points.

4.1.1 Scala is a young language

In the one hand, this means that Scala is not stable yet. There are still some bugs in the compiler and in the specifications. On the other hand, the community is very active. When the bugs are major they are fixed quickly (in the nightly builds) and otherwise they enter in the next release. Moreover there are very active mailing lists and when there are doubts or question about how to do something, and so on, the problems are solved, usually, within the day.

4.1.2 Scala is easy to learn

There is some good documentation available freely on the Scala website ([scala]) and a book is available for buying ([MOV08]). I did not have the opportunity to see it but it seems, by the echoes I had, very good and very complete. What’s more, for people coming from Java, there are some short papers to learn Scala quickly (for instance [SH08]). Of course having a previous experience of functional programming is a plus as it allows to enter more easily into Scala functional way of thinking.

4.1.3 Scala is object-oriented and functional

I did not have any previous experience of a language that mixed both paradigms but in my sense, the Scala integration is successful. It allows to express things in a light way, and to take advantage of the two paradigms. The oriented object aspect allows to create classes and to have a great type system. The functional one allows pattern matching and work on lists that helps to write things in a light way. With some practice it is easy to mix both aspects. However, if one does not do this effort, there are few advantages to go, for example, from Java to Scala. For my part, I noticed that the code I was writing was light and short in general. Moreover, the functional programming aspect of Scala makes programs elegant and synthetic.

4.1.4 Multiple inheritance

As said in 3.2.1, Scala does not offer a true mechanism for multiple inheritance. Traits can be a good compromise, however they make the code cumbersome. But this remains reasonable as traits are only created when needed. For the whole AADL metamodel (around 200 classes), 35 additional traits are created, which is not that much. However, this remains a stopgap.
4.1.5 Mild-efficiency

For big programs, Scala is quite efficient (for the Fiacre example, from metamodel parsing to model pretty printing, it lasts 1:18 minute), but there is an un-compressible compilation time no matter how many lines the program has. But this is mainly annoying for small programs.

4.1.6 Pattern matching

Pattern matching was one of the things that let think that Scala could be interesting. And indeed, it allowed one to express a lot of things with a nice syntax. However, when the compiler finds a \texttt{match/case} instruction, it converts everything to casts. It only ensures that the cast is correct. This disappointed me but otherwise, the syntax is light and easy to use.

4.1.7 Global conclusions

Even if Scala disappointed us on some points, it presents some clear interest in comparison with Java and is worth studying. It has strong theoretic background and for that reason has interesting features. For my part, I really enjoyed programming with Scala as it offered a very good mix in my sense, of oriented-object and functional programming.

4.2 Comparison of the result with the initial objectives of the project

The initial global objective of the project was to make a general framework for model transformation that given a metamodel $A$, a metamodel $B$ and the transformation from $A$ to $B$ would be able to transform any model of $A$ into a model of $B$. The initial concrete application was to realize the transformation from AADL to Fiacre.

4.2.1 Change in the objective

As we said in chapter 3.4, the objectives concerning the final application changed during the internship. Instead of a transformation from AADL to Fiacre, a transformation from Fiacre to Fiacre was done. As we said, this second objective was useful too and completed.

4.2.2 The framework

Apart from the special features of the AADL metamodel, the objective to realize a framework for model transformation was completed. The application can take any two metamodels, with not too special features (see annex C for supported features), and given the transformation from the one to the other can transform any model of the first metamodel into a model of the second one. What’s more, I tried as much as possible to make the framework extensible so that someone else than me could make it support new features. By the way, I will spend a good amount of the remaining time making documentation so that entering in the code will not be too hard. In the same way, all the code has been commented in order to generate documentation with Scaladoc (see appendix E). Moreover, a user manual (see appendix D) and a technical manual have been delivered.

4.2.3 Conclusions about Scala

Another objective of the project was to evaluate the Scala language. The conclusions I could get after a 6 months practice are explained in chapter 4.1.
4.3 What remains to be done

A six months internship was too short for me to do all what I would have liked. There is still a lot to do but I miss time. Here is a list of improvements and extensions for the current application. We will not repeat the points concerning the AADL metamodel listed in the annex C.

- One not completed objective was the pluginisation of the application. This meant first, to make the application itself a plugin of Topcased, for it to be easy to use; second, to generate automatically a plugin when parsing a metamodel that would realize model parsing, transformation and pretty printing. This would probably have encouraged people to use Scaltrans.

- I believe the code generation for model parser and pretty printer could be improved. For now it is mainly strings printing. Although some general functions were made for the code not to be too repetitive (e.g. functions printing match/cases functions), I think a higher abstraction level is reachable. However, the current solution result is acceptable in shape and efficiency.

- It could be useful to offer more features in the model transformation framework. For instance, a function returning all the transformations applicable to a model, letting the user choose the one to use. Or a transformation function taking the object as parameter instead of its attributes.

- It could also be interesting to insert the recurrency on the transformation call into the framework instead of making the user write it. And to improve the mechanism, as for parsing and pretty printing, so the model is read only once, the information stored and the image created at the end. Maybe it could be worth trying to make the framework inherit from T2U, even if I suspect it would involve a great loss in efficiency.

- I would have found interesting for closure to bootstrap. Indeed, the metamodel parsing can be seen as a transformation from an instance of the Ecore metamodel to an instance of the Scala metamodel. It should be possible to make a metamodel parser that way (see figure 4.1): parse the Ecore metamodels, get the needed files (the parser and the model transformation framework generated and use as pretty printer the functions of the Scala metamodel that print Scala code), write the transformation from the Ecore to Scala (using the transformation framework) and add the code generator for model parsing, transformation and pretty printing.

![Figure 4.1: Bootstrap of the application](image)

Nevertheless, this would certainly imply a loss in efficiency as the operations of parsing and translating to Scala are done together in the parser. That’s why it was not a priority.

- Finally, it could be useful to accept the input metamodel in Java as an alternative to Ecore, and then translate it to Scala. Sometimes the metamodel does not exist under an Ecore format and it could be good to be able to use the application directly with the metamodel in Java. This seemed a definitely useful improvement but it was suggested very late in the internship (mid-july). However, if the idea about bootstrapping mentioned above actually works, a first implementation of this suggestion could maybe be created easily, doing the above steps, but replacing the Ecore metamodel by the Java metamodel. However, one question that is not solved yet is the way to read Java code the same way XML code can be read (with scala.xml).

There is still a lot to improve. Nevertheless, the current version can be considered as stable and acceptable. For now, effort has to be put on the documentation aspect for someone else to be able to improve the existing application with new features, as for example, the above mentioned ones.
4.4 Planning

At the beginning of the project, the following planification was made.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Week 1</th>
<th>Week 2</th>
<th>Week 3</th>
<th>Week 4</th>
<th>Week 5</th>
<th>Week 6</th>
<th>Week 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formation on Automation &amp; Expressions example</td>
<td>02/18-02/22</td>
<td>02/19-02/22</td>
<td>02/26-03/02</td>
<td>03/06-03/14</td>
<td>03/17-03/29</td>
<td>03/25-03/30</td>
<td>03/31-04/04</td>
</tr>
<tr>
<td>Core to Scala</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application on AADL to Fiacre</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Benchmarking AADL to Fiacre / Kermeta</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If we look at it, we see that there were twelve weeks of development and tests (weeks 8 to 19). This was pretty much respected even if the parts of tests and development were not as well sepaerated as in the planification. First the Fiacre metamodel was introduced and then the AADL metamodel. They were introduced step by step and the testing on it was constant. The documentation has really begun at beginnings of July and for the moment is not over. It will last at least two more weeks. For the rest I cannot say anything yet.

4.5 Personal conclusions

This intership was for me the occasion to discover how such a big company as the CNES works. It was interesting to talk with the agents, to have their point of view and thoughts on the working world.

It also allowed me to discover the functioning of a big project such as Topcased. I could interact with various actors of this project of different kinds (the CNES, a laboratory (IRIT), an ITSP (Anyware Technologies)). That let me see a lot of different faces and aspects. Moreover, I had the opportunity to assist to various conferences and reunions around the Topcased project (or not) and I learned a lot during these. I appreciated a lot to be able to meet various actors of the project and to work with them. This way, I could have very varied points of views and opinions and I really believe it helped me for this intership.

On a more technical aspect, I really have the feeling that I learned a lot. First, of course, I had to learn a new language (even though it was not too far from what I already knew) and I had to use Eclipse of which I did not even suspect the existence before this intership. I also learned to make scaladoc (very close to javadoc) which can always be useful. But on a more scientific aspect, I could understand better some rules to write code and practice some software engineering concepts that I had learned at school but that I never had the opportunity to test.

Working iteratively helped me to understand my conceptions errors. Indeed, I released a version (or a partial version) and my tutor would indicate and explain me the points to be improved or
changed. This leaded me to a quite coherent architecture. This also allowed me to get each time to a higher level of abstraction in the final idea to arrive to something really general. However, I have to say that the main work of this internship was the application design (write code) and test. More than what I thought. But I liked it anyway because it made me learn a lot. Finally, I appreciated the links this project had with the research world as I will start a PHD from next october.
Bibliography


Appendix
Appendix A

Model transformation framework for the expressions example

```scala
package etestCelia

trait Signature {
  type TargetProduit <: TargetOperation
  type TargetSomme <: TargetOperation
  type TargetOperation <: TargetExp
  type TargetVarRef <: TargetExp
  type TargetVarDecl <: TargetExp
  type TargetNum <: TargetExp

  def transformProduit ( arg1 : TargetExp , arg2 : TargetExp ) : TargetProduit
  def transformSomme ( arg1 : TargetExp , arg2 : TargetExp ) : TargetSomme
  def transformVarRef ( decl : TargetVarDecl ) : TargetVarRef
  def transformVarDecl (name : Predef.String , body : TargetExp ) : TargetVarDecl
  def transformNum (n : scala.Int ) : TargetNum
}

class Iterator(val s: Signature) {
  import scala.collection.mutable.HashMap
  var otherPass = false

  def findObjInHm[T, U](hm: HashMap[T, U])(x: T): U = {
    val res = hm.get(x)
    if (res == None) { otherPass = true
      res.getOrElse(null).asInstanceOf[U]
      true
    }
  }

  val hmVarDecl : HashMap[VarDecl, s.TargetVarDecl] = new HashMap()

  def iterProduit(x: Produit): s.TargetProduit =
    s.transformProduit(iterExp(x.arg1), iterExp(x.arg2))
  def iterSomme(x: Somme): s.TargetSomme =
    s.transformSomme(iterExp(x.arg1), iterExp(x.arg2))
  def iterVarRef(x: VarRef): s.TargetVarRef =
    s.transformVarRef(findObjInHm[VarDecl, s.TargetVarDecl](hmVarDecl)(x.decl))
  def iterVarDecl(x: VarDecl): s.TargetVarDecl =
    val trans = s.transformVarDecl(x.name, iterExp(x.body))
    hmVarDecl.update(x, trans)
    trans
  
  def iterExp(x: Exp): s.TargetExp = x match {
    case y : VarDecl =>
      val trans = s.transformVarDecl(y.name, iterExp(y.body))
      hmVarDecl.update(y, trans)
      trans
    case VarRef(decl) => s.transformVarRef(findObjInHm[VarDecl, s.TargetVarDecl](hmVarDecl)(decl))
    case Num(n) => s.transformNum(n)
    case Produit(arg1, arg2) => s.transformProduit(iterExp(arg1), iterExp(arg2))
    case Somme(arg1, arg2) => s.transformSomme(iterExp(arg1), iterExp(arg2))
    case _ => error("...")
  }
}
```
Listing A.1: Generated model transformation framework for the Expressions example
Appendix B

Fiacre Metamodel

<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  nsURI="http://org.topcased.fiacre" nsPrefix="fiacre">
  <eClassifiers xsi:type="ecore:EClass" name="Program">
    <eStructuralFeatures xsi:type="ecore:EReference" name="declarations" lowerBound="1" upperBound="-1" eType="#//Declaration" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="root" lowerBound="1" eType="#//NodeDecl"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Declaration" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="NodeDecl" abstract="true">
    <eSuperTypes="#//Declaration"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="args" upperBound="-1" eType="#//ArgumentVariable" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="vars" upperBound="-1" eType="#//LocalVariable" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ports" upperBound="-1" eType="#//ParamPortDecl" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="localPorts" upperBound="-1" eType="#//LocalPortDecl" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="TypeDecl">
    <eSuperTypes="#//Declaration"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="is" lowerBound="1" eType="#//Type" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ChannelDecl">
    <eSuperTypes="#//Declaration"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="is" lowerBound="1" eType="#//Channel" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ComponentDecl">
    <eSuperTypes="#//NodeDecl"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="body" lowerBound="1" eType="#//Composition" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="priority" upperBound="-1" eType="#//Priority" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ProcessDecl">
    <eSuperTypes="#//NodeDecl"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="states" upperBound="-1" eType="#//State" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="transitions" upperBound="-1" eType="#//Transition" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="initAction" lowerBound="1" eType="#//Statement" containment="true"/>
  </eClassifiers>
</ecore:EPackage>
APPENDIX B. FIACRE METAMODEL

```xml
<eClassifiers xsi:type="ecore:EClass" name="PortDecl" eSuperTypes="#//PortDecl"/>
<eClassifiers xsi:type="ecore:EClass" name="WhileStmt" eSuperTypes="#//Statement">
<eStructuralFeatures xsi:type="ecore:EReference" name="condition" lowerBound="1" eType="#//Exp" containment="true"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="body" lowerBound="1" eType="#//Statement" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="IfStmt" eSuperTypes="#//Statement">
<eStructuralFeatures xsi:type="ecore:EReference" name="condition" lowerBound="1" eType="#//Exp" containment="true"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="then" lowerBound="1" eType="#//Statement" containment="true"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="else" lowerBound="1" eType="#//Statement" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Select" eSuperTypes="#//Statement">
<eStructuralFeatures xsi:type="ecore:EReference" name="statements" lowerBound="1" upperBound="-1" eType="#//Statement" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="To" eSuperTypes="#//Statement"/>
<eClassifiers xsi:type="ecore:EClass" name="Statement" abstract="true"/>
<eClassifiers xsi:type="ecore:EClass" name="DeterministicAssignment" eSuperTypes="#//Assignment">
<eStructuralFeatures xsi:type="ecore:EReference" name="assignments" lowerBound="1" upperBound="-1" eType="#//SingleAssignment" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="NonDeterministicAssignment" eSuperTypes="#//Assignment">
<eStructuralFeatures xsi:type="ecore:EReference" name="lhs" lowerBound="1" upperBound="-1" eType="#//Pattern" containment="true"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="condition" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="SingleAssignment">
<eStructuralFeatures xsi:type="ecore:EReference" name="lhs" lowerBound="1" eType="#//Pattern" containment="true"/>
<eStructuralFeatures xsi:type="ecore:EReference" name="rhs" lowerBound="1" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="UnExp" eSuperTypes="#//Exp">
<eStructuralFeatures xsi:type="ecore:EReference" name="exp" lowerBound="1" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="BinExp" eSuperTypes="#//Exp">
<eStructuralFeatures xsi:type="ecore:EReference" name="exp" lowerBound="1" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="Unop">
<eLiterals name="UMINUS"/>
<eLiterals name="UDOLLAR" value="1"/>
<eLiterals name="UNOT" value="2"/>
<eLiterals name="UFULL" value="3"/>
<eLiterals name="UEMPTY" value="4"/>
<eLiterals name="DEQUEUE" value="5"/>
<eLiterals name="FIRST" value="6"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EEnum" name="BinOp">
<eLiterals name="BOR" literal="BOR"/>
<eLiterals name="BAND" value="1" literal="BAND"/>
```
<eStructuralFeatures xsi:type="ecore:EReference" name="decl" lowerBound="1" eType="#//TypeDecl"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Channel" abstract="true"/>
<eClassifiers xsi:type="ecore:EClass" name="ChannelId" eSuperTypes="#//Channel">
  <eStructuralFeatures xsi:type="ecore:EReference" name="decl" lowerBound="1" eType="#//ChannelDecl"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Profile" eSuperTypes="#//Channel">
  <eStructuralFeatures xsi:type="ecore:EReference" name="types" upperBound="-1" eType="#//Type" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="OrChannel" eSuperTypes="#//Channel">
  <eStructuralFeatures xsi:type="ecore:EReference" name="lhs" lowerBound="1" eType="#//Channel" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="rhs" lowerBound="1" eType="#//Channel" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="InlineArray" eSuperTypes="#//InlineCollection"/>
<eClassifiers xsi:type="ecore:EClass" name="InlineRecord" eSuperTypes="#//Exp">
  <eStructuralFeatures xsi:type="ecore:EReference" name="values" lowerBound="1" upperBound="-1" eType="#//ValuedField" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ValuedField">
  <eStructuralFeatures xsi:type="ecore:EReference" name="value" lowerBound="1" eType="#//Exp" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="field" lowerBound="1" eType="#//Field"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Synchronization" eSuperTypes="#//Communication"/>
<eClassifiers xsi:type="ecore:EClass" name="Reception" eSuperTypes="#//Communication">
  <eStructuralFeatures xsi:type="ecore:EReference" name="pattern" lowerBound="1" upperBound="-1" eType="#//Pattern" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="where" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Emission" eSuperTypes="#//Communication">
  <eStructuralFeatures xsi:type="ecore:EReference" name="args" upperBound="-1" eType="#//Exp" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Seq" eSuperTypes="#//Statement">
  <eStructuralFeatures xsi:type="ecore:EReference" name="statements" upperBound="-1" eType="#//Statement" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="LocalPortDecl" eSuperTypes="#//PortDecl">
  <eStructuralFeatures xsi:type="ecore:EReference" name="mini" lowerBound="1" eType="#//MinBound" containment="true"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="maxi" lowerBound="1" eType="#//MaxBound" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ParamPortDecl" eSuperTypes="#//PortDecl"/>
<eClassifiers xsi:type="ecore:EClass" name="Variable" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#/EString"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="type" lowerBound="1" eType="#//Type" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Priority">
  <eStructuralFeatures xsi:type="ecore:EReference" name="inf" lowerBound="1" upperBound="-1" eType="#//PortDecl"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="sup" lowerBound="1" upperBound="-1" eType="#//PortDecl"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ConstrExp" eSuperTypes="#//Exp">
  <eStructuralFeatures xsi:type="ecore:EReference" name="arg" eType="#//Exp"
Listing B.1: Fiacre metamodel
Appendix C

Managed and unmanaged features

C.1 What is done

C.1.1 Steps of the application

- design (in Scala) of the Scala metamodel
- generation of an helper file, with default values
- reading of an Ecore file.
- creation of the associate Scala structures (in the Scala metamodel)
- pretty printing of the Scala structures (according to the scala syntax)
- parsing of a model (xmi) according to the metamodel
- creation of the associated Scala objects
- pretty printing (to xmi) of Scala objects

C.1.2 Managed Scala structures

Packages
A package may have:
- a name
- a upper package in which it is included
- inner classes attached to it
- a list of classes to import

Classes
A class may have:
- a name
- a list of attributes
- a list of super classes (keeping in mind that in Scala, multiple inheritance is not allowed. Only mixins are possible and in this case the classes must be traits)

A class may be (apart from a classical class):
- a case class
• an object
• a case object
• an abstract class
• an abstract case class
• a trait

A class may also be an enumeration (actually, an enumeration is a hierarchy of classes). An enumeration may have:

• a name
• a list of literals (each literal have a name, and may have a value)

An enumeration will be traduced to Scala as an case class of name the name of the enumeration, no super class and two attributes (name and value). Each literal will be traduced as a case object inheriting from the case class, with attribute name set to the name of the literal and attribute value set to the value of the literal (0 if no value)

Attributes
An attribute may have:

• a name
• a type
• a "kind" of type (list, option, and so on)
• bounds (a lower and an upper bound)

We can also indicate whether the attribute must be a constructor or not
An attribute may be:

• a reference attribute (reference to another object)
• a class attribute (containment in the class)
• a data attribute (of a data type)

Data types
A data type may be:

• a Double
• an Integer
• a Short
• a Long
• a Float
• a String
• a Boolean
C.1.3 Managed properties of the Ecore file

The different types of objects recognized and handled by the application currently are:

- EPackage
- EClassifiers - the types of classifiers handled are:
  - EClass
  - EDataType
  - EEnum
- EStructuralFeature

EPackage

The properties of an EPackage handled by the application are:

- the attribute "name"
- the children (which are usually EClassifiers)

EClass

The properties of an EClass handled by the application are:

- the attribute "name"
- the attribute "abstract"
- the attribute "interface" (\rightarrow trait)
- the attribute "eSuperTypes" which corresponds to the super class of this class. There can be various super classes and they can be located in different files.
- the children that will be interpreted as attributes of the class. For the moment only children of type EStructuralFeatures are handled.
- children (which are usually EClassifiers)

EDataType

The EDataTypes recognized by the application are:

- EInt
- EShort
- ELong
- EFloat
- EDouble
- EBoolean
- EString

EEnum

The properties of an EEnum handled by the application are:

- the attribute "name"
- the children "eLiterals" (each literal has a name and may have a value)
**EStructuralFeatures**

The properties of an EStructuralFeatures handled by the application are:

- the attribute "name"
- the attribute "upperBound"
- the attribute "lowerBound"
- the attribute 'xsi:type' ; it can have 2 values: 'EReference' or 'EAttribute' and allows to identify if the attribute will be traduced to Scala as a Reference Attribute (EReference), a Class Attribute (EReference) or a Data Attribute (EDataType).
- the attribute "containment" ; it allows to choose, for an attribute of type 'EReference' between a Reference Attribute (containment = false) and a Class Attribute (containment = true)
- the attribute 'eType' ; it corresponds to the type of the attribute. The attribute can be of type EDataType, EEnum or EClass. In these two last cases, the EEnum or EClass may be located in a different file. This property is also linked to the attribute 'xsi:type': if xsi:type = 'EReference' then eType is an Eclass, otherwise, eType may be an EDataType or an EEnum

**C.1.4 Managed properties of the helper file**

**For an EPackage**

For now, no special property can be set for an EPackage in the helper file. However, maybe the name of the upper package (the package to which this one belongs) should appear here.

**For an EClass**

For now the properties of an EClass that can be set thanks to the helper file indicate whether:

- the class should be traduced as a case class
- the class should be traduced as an object
- the class should be considered as the super class for all the other classes of the metamodel (but this might have a very poor interest and should maybe be removed)

**For an EEnum**

For now, no special property can be set for an EEnum in the helper file.

**For an EDataType**

For now, no special property can be set for an EDataType in the helper file.

**For an EStructuralFeatures**

For now, no special property can be set for an EStructuralFeatures in the helper file.

**C.2 What remains to be done**

**C.2.1 Identified unmanaged Scala structures**

**Data types**

For the moment, a data type may not be:

- the data type corresponding to an 'EFeatureMapEntry'
Operations
For the moment, no managing of operation attached to classes is done. However I doubt it really is interesting as there can’t be any operation in a meta model.

C.2.2 Identified unmanaged properties of the Ecore file

EClass
For the moment the properties of an EClass that are not handled are:

- the children of type "EAnnotations"
- Scala does not accept classes with more than 22 attributes so I will have to see how to handle them

EDataTypes
For the moment, the EDataTypes that are not handled are:

- EFeatureMapEntry

EStructuralFeatures
For the moment the properties of an EStructuralFeatures that are not handled are:

- the children of type "EAnnotations"
- the attribute "transient" (I don’t know what it means)
- the attribute "unique"
- the attribute "unsettable"
- the attribute "volatile"
- the attribute "derived" (corresponds to volatile = true, transient = true and changeable = false)
- the attribute "required" (means that lowerBound > 0) ; this attribute is of a low priority as it does not appear in any of the AADL metamodel files nor in the Fiacre metamodel.
- the attribute "eOpposite"
- the attribute "resolveProxies"
- the attribute "defaultValueLiteral" (this one seems to appear only in case the attribute is of type EEnum or if it is an EBoolean (which is an enumerated type), and it seems that it is the default value the attribute should take)
- the attribute "many" (means that upperBound > 1) ; this attribute is of a low priority as it does not appear in any of the AADL metamodel files nor in the Fiacre metamodel.
- the attribute "changeable" ; this attribute is of a low priority as it does not appear in any of the AADL metamodel files nor in the Fiacre metamodel.
EAnnotations

For the moment, EAnnotations are not handled at all. However we can remark that:

- if the EAnnotations is embedded in an EClass, its attributes may be:
  - isProxy and usually has value 'true'

- if the EAnnotations is embedded in an EStructuralFeatures, its attributes may be:
  - if the type of the attribute is "EFeatureMapEntry":
    - kind and usually has value 'group'
    - namespace
  - otherwise:
    - group

Note: theses remarks are only based on the observation of the Ecore files of the AADL metamodel
Scaltrans

Model transformation with Scala

User’s Manual
Introduction

The objective of this manual is to explain the way the application Scaltrans should be used. This application aims at offering a general framework for model transformation. The idea is to be able to transform any model instance of a metamodel A into a model instance of a metamodel B, as long as we dispose of the two metamodels and of the transformation rules to go from A to B.

In a first part we will see how to do to parse metamodels. Then we will see how to write model transformation rules, how to parse and pretty print models.
## Contents

1 How to parse metamodels ?  
   1.1 The metamodel file .......................... 5  
   1.2 The configuration file .......................... 6  
   1.3 Metamodel parser .......................... 7  

2 How to parse, transform and pretty print models ?  
   2.1 Model parsing .................................. 8  
   2.2 Model transformation .......................... 8  
      2.2.1 Transformation definition .................. 8  
      2.2.2 Transformation usage ...................... 10  
   2.3 Model pretty printing .......................... 10  
   2.4 Complete file for model treatment .................. 10
Chapter 1

How to parse metamodels?

The first thing to do when using Scaltrans is to parse metamodels. To do so, you will need various things:

- a metamodel file
- a configuration file
- directions of files where to store the generated code

1.1 The metamodel file

The metamodel needs to be under Ecore format (see). It can be divided into various files. In the rest of this manual, we will use as an example to illustrate our matter the example of a simple expressions grammar. Figure 1.1 and listing 1.1 show the schema of the metamodel of this example and its Ecore translation.

Figure 1.1: Metamodel of the Expressions example

```xml
<?xml version="1.0" encoding="UTF-8"?>
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    name="expExample" nsURI="http://expExample" nsPrefix="expExample">
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="body" eType="expExample:Exp"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="expr1" eType="expExample:Exp"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="expr2" eType="expExample:Exp"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="num" eType="expExample:Num"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="varDecl" eType="expExample:VarDecl"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="varRef" eType="expExample:VarRef"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="operation" eType="expExample:Operation"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="somme" eType="expExample:Somme"/>
    <eClassifiers xsi:type="ecore:EStructuralFeature" name="produit" eType="expExample:Produit"/>
</ecore:EPackage>
```
1.2 The configuration file

To create the appropriate structures, the application needs configuration files (one per file of the metamodel). You can create it with the function: createHelperFile of the package HelperFileCreator. To call this function you have to write:

```
HelperFileCreator.createHelperFile([path]/metamodel_file_1, [path]/config_file_1)
HelperFileCreator.createHelperFile([path]/metamodel_file_2, [path]/config_file_2)
...  
HelperFileCreator.createHelperFile([path]/metamodel_file_n, [path]/config_file_n)
```

Then, in the files config_file_1, ..., config_file_n you will have the configuration files corresponding to the different files of the metamodel. For the expressions example, the generated file is shown in listing 1.2.

```
<xml version="1.0" encoding="UTF-8"/>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmllns:xsi="http://www.w3.org/2001/XMLSchema-instance"
nsURL="http://expExample" nsPrefix="expExample">
<eClassifiers xsi:type="ecore:EClass" name="Exp" isSuperClass="false">
<eStructuralFeatures xsi:type="ecore:EAttribute" name="n" lowerBound="1" />
<eStructuralFeatures xsi:type="ecore:EAttribute" name="body" lowerBound="1" />
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Num" isCaseClass="true">
<eStructuralFeatures xsi:type="ecore:EAttribute" name="n" />
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="VarDecl" isCaseClass="true">
<eStructuralFeatures xsi:type="ecore:EAttribute" name="name" />
<eStructuralFeatures xsi:type="ecore:EReference" name="body" lowerBound="1" containment="true" />
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="VarRef" isCaseClass="true">
<eStructuralFeatures xsi:type="ecore:EReference" name="decl" lowerBound="1" />
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Operation" abstract="true" isSuperClass="false">
<eStructuralFeatures xsi:type="ecore:EReference" name="arg1" lowerBound="1" containment="true" />
<eStructuralFeatures xsi:type="ecore:EReference" name="arg2" lowerBound="1" containment="true" />
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Somme" isCaseClass="true">
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Produit" isCaseClass="true">
</eClassifiers>
</ecore:EPackage>
```
Listing 1.2: Configuration file for the expressions example

As you can see, each class is reproduced with its name and its attributes (**eStructuralFeatures**). What’s more, some additional attributes can be added to the **eClassifiers** to indicate what kind of object should be created. The different options that exist are:

- **isCaseClass**: if set to true, indicates that the class to create is a case class
- **isObject**: if set to true, indicates that the class to create is an object

By default, these options are set to false. You can set the value of these attributes by modifying the configuration file.

Only the classes have special features set in the configuration file.

### 1.3 Metamodel parser

Now that we have all the elements, we can parse the metamodel. For this purpose, you will need to specify files directions where to store the files generated during parsing. For each metamodel file, there are four files that are generated: Scala code corresponding to the metamodel, model parser, model transformation framework, model parser. We will call them respectively GMMi, GMMiParser, GMMiTransfo and GMMiPrettyPrint, where *i* corresponds to the metamodel file number.

To parse the metamodel, you will have to call **parser** of package **WholeMM2Scala** in the following way:

```scala
WholeMM2Scala.parser(  
    global_package_name,  
    List(  
      ([path]/config_file_1.ecore, [path]/metamodel_file_1, [path]/GMM1.scala),  
      ([path]/config_file_2.ecore, [path]/metamodel_file_2, [path]/GMM2.scala),  
      ...  
      ([path]/config_file_n.ecore, [path]/metamodel_file_n, [path]/GMMn.scala)  
    ),  
    List(  
      (GMM1Parser.scala, GMM1PrettyPrint.scala, GMM1Transfo.scala),  
      (GMM2Parser.scala, GMM2PrettyPrint.scala, GMM2Transfo.scala),  
      ...  
      (GMMnParser.scala, GMMnPrettyPrint.scala, GMMnTransfo.scala)  
    )  
)
```

As a concrete example, listing 1.3 shows the whole content of the file used to parse the metamodel of the expressions example.

```scala
object MMExpParsing {  
  import org.topcased.scalatrans.mmparser.WholeMM2Scala  
  def main(args: Array[String]) : Unit = {  
    var args = List(('configExp.ecore','mmexp.ecore','Exp.scala'))  
    var files = List(('ExpParser.scala','ExpPP.scala','ExpTransfo.scala'))  
    WholeMM2Scala.parser('', args, files)  
  }  
}
```

Listing 1.3: Code for metamodel parsing for the expressions example
Chapter 2

How to parse, transform and pretty print models?

The very first thing to do, when willing to deal with model, is to compile the files generated during metamodel parsing in order to be able to use them. Then, there are three phases. We have to parse models, write the transformation rules and use them, and pretty print the resulting model.

In this section, I suppose you dispose of a model. If you don’t know how to create one, you can refer to the Topcased documentation and tutorials available at http://www.topcased.org/.

2.1 Model parsing

The first thing to do is to transform the XMI model into Scala objects. To do so, you will have to use the function parserFromFile of the class MMiParser created during metamodel parsing. MMi must be the file of the metamodel that defines the class of the object (model) you want to parse. In concrete terms, what you will have to write is:

```scala
val parser = new [global_package_name.]MMiParser
val parsedModel = parser.parserFromFile(List([path]/model_file))
```

parsedModel now contains the Scala object corresponding to the model contained in model_file.

2.2 Model transformation

Here is the earth of the application as this is where we will define the model transformation rules to achieve the model transformation the application aims at.

2.2.1 Transformation definition

To define the transformation there are 2 different cases.

The general case

The way of proceeding that we will explain now can be used for any cases, even when the method explained in next section could be applied.

To define a new transformation, you have to implement each abstract type and each transformXX function defined in the trait Signature in each file GMMiTansfo. The abstract type represent the type of the transformed object, for each class defined in the input metamodel. The functions transformXX indicate for each class on the input metamodel, how to transform it into an object of the output metamodel. What’s more, you will have to define an object that extends Iterator and
that takes as attribute the previously defined transformation. When willing to transform a model, 
you will have to call the functions of this new Iterator. If we call A the input metamodel, and B 
the output metamodel, here is what you should write for each file of the metamodel.

```scala
import global_package_name.MMi._
object TransfoMMi extends Signature {
  type TargetCA1 = CBj1
  ...
  type TargetCA2 = CBj2
  ...
  type TargetCAn = CBjn

  override def transformCA1(...) = CBj1(...)
  override def transformCA2(...) = CBj2(...)
  ...
  override def transformCAn(...) = CBjn(...)
}
object IterTransfoMMi extends Iterator(TransfoMMi)
```

As a concrete illustration, from our expressions example, if we want to transform a Somme into a
Produit and inverse the terms of a Produit, listing 2.1 shows what we would have to write.

```scala
object TransfoSomme extends Signature {
  type TargetProduit = Produit
  type TargetSomme = Produit
  type TargetOperation = Operation
  type TargetVarRef = VarRef
  type TargetVarDecl = VarDecl
  type TargetNum = Num
  type TargetExp = Exp

  override def transformProduit(arg1:TargetExp, arg2:TargetExp):TargetProduit = Produit(arg2, arg1)
  override def transformSomme(arg1:TargetExp, arg2:TargetExp):TargetSomme = Produit(arg1, arg2)
  override def transformVarRef(decl:TargetVarDecl):TargetVarRef = VarRef(decl)
  override def transformVarDecl(name:Predef.String, body:TargetExp):TargetVarDecl = VarDecl(name, body)
  override def transformNum(n:scala.Int):TargetNum = Num(n)
}
object IterTransfoSomme extends Iterator(TransfoSomme)
```

Listing 2.1: A concrete example: transformation of a Somme into a Produit and inversion of the
terms of a Produit

A very restrictive case

We will now explain here a very particular case.

As you can have seen above, if the metamodels are long, write a new transformation can be quite
tedious. There is one case in which this can be lightened. This happens when the input metamodel
and the output metamodel are the same (like in listing 2.1 AND when the target types are equals
to their source types. Actually, that means that the only thing that change in regard with the
Identity transformation are the transformXX functions. In that case, you can inherit from the
Identity class and only redefine the transformXX functions you need.

In concrete terms you will write something like this:

```scala
import global_package_name.MMi._
object TransfoMMi extends Identity {
  override def transformCAj1(...) = CAk1(...)
  override def transformCAj2(...) = CBk2(...)
  ...
  override def transformCAjm(...) = CBkm(...)
}
```
object IterTransfoMMi extends Iterator(TransfoMMi)

As a concrete example, still in our expressions example, we could want to inverse the terms of a Somme. In that case would write the code shown in listing 2.2.

object TransfoInverseSomme extends Identity {
  override def transformSomme(arg1: TargetExp, arg2: TargetExp): TargetSomme = 
  Somme(arg2, arg1)
}

object IterTransfoInverseSomme extends Iterator(TransfoInverseSomme)

Listing 2.2: A concrete example: inversion of a Somme terms

2.2.2 Transformation usage

Now that we have written the transformation we want to realize (they can be written in separate files), we have to use them. We will call IterTransfoMM the iterator of the transformation we have write, and parsedModel the Scala object containing the parsed model. In order to be able to parse the model, you will have to know its type. We will call it CAi here.

To use the transformation, you have to call the iterCAi function of IterTransfoMM. You might have to iterate this transformation various times if you want to have a completely transformed model. This only applies in case the output metamodel is the same as the input metamodel.

Let’s take an example. In the example shown in listing 2.1 if the object we are transforming is a Somme there are actually two transformations to realize: first transform it to a Produit and then inverse the terms of the Somme. To realize these two steps, an iteration is necessary.

The code below shows how to do in order to realize this iteration on the expressions example:

```scala
var transformedModel = IterTransfoSomme.iterExp(parsedModel.asInstanceOf[Exp])
while (IterTransfoSomme.otherPass) {
  IterTransfoSomme.otherPass_=(false)
  transformedModel = IterTransfoSomme.iterExp(ep.asInstanceOf[Exp])
}
```

But in general, the code to write will be

```scala
var transformedModel = IterTransfoMM.iterCAi(parsedModel.asInstanceOf[CAi])
```

At the end of these operations, transformedModel will contain the transformed model.

2.3 Model pretty printing

Now, the only thing that remains to be done is to transform the Scala object contained in transformedModel to XMI. To do so you will have to use the function prettyPrint of the package MMiPrettyPrint. In concrete terms, what you will have to write is:

```scala
val prettyPrinter = new [global_package_name.]MMiPrettyPrint
val prettyPrintedModel = prettyPrinter.prettyPrint(transformedModel)
```

prettyPrintedModel now contains the XMI code corresponding to the transformed model. It can be printed on the screen, or to a file.

2.4 Complete file for model treatment

To sum up what we have said for model treatment, here is the code that has to be written in order to do a model transformation in our expressions example, with the transformation IterTransfoSomme.
object ModelExpParsing {
  import expExample._
  val parser = new MMExpParser
  val parsedModel = parser.parserFromFile(List(model_file))
  var transformedModel = IterTransfoSomme.iterExp(parsedModel.asInstanceOf[Exp])
  while (IterTransfoSomme.otherPass) {
    IterTransfoSomme.otherPass_=(false)
    transformedModel = IterTransfoSomme.iterExp(ep.asInstanceOf[Exp])
  }
  val prettyPrinter = new [global_package_name.]MMExpPrettyPrint
  val prettyPrintedModel = prettyPrinter.prettyPrint(transformedModel)
}

object ModelExpParsing {
  import expExample._
  val parser = new MMExpParser
  val parsedModel = parser.parserFromFile(List(model_file))
  var transformedModel = IterTransfoSomme.iterExp(parsedModel.asInstanceOf[Exp])
  while (IterTransfoSomme.otherPass) {
    IterTransfoSomme.otherPass_=(false)
    transformedModel = IterTransfoSomme.iterExp(ep.asInstanceOf[Exp])
  }
  val prettyPrinter = new [global_package_name.]MMExpPrettyPrint
  val prettyPrintedModel = prettyPrinter.prettyPrint(transformedModel)
}
Appendix E
Extract of the scaladoc generated for the application
object GenericFunctions extends AnyRef

Object that contains some generic functions used in various classes of the package.

Method Summary

<table>
<thead>
<tr>
<th>def</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method that reads a file (XML) and creates the corresponding node</td>
<td></td>
</tr>
<tr>
<td>def printToFile (filename : java.lang.String, toWrite : java.lang.String) : Unit</td>
<td></td>
</tr>
<tr>
<td>Method that prints to a file</td>
<td></td>
</tr>
<tr>
<td>Method that transform a string into a list of strings, cutting it at each white space.</td>
<td></td>
</tr>
</tbody>
</table>

Methods inherited from AnyRef

getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from Any

==, !=, isInstanceOf, asInstanceOf

Method Details

def printToFile(filename : java.lang.String, toWrite : java.lang.String) : Unit

Method that prints to a file

Parameters

filename - the name of the file where we want to print
toWrite - the string to write in the file


Method that reads a file (XML) and creates the corresponding node

Parameters

filename - the name of the file we want to parse

Returns
The created node

```scala
```

Method that transform a string into a list of strings, cutting it at each white space.

**Parameters**
- `str` – the string we want to transform into a list

**Returns**
The created list of strings

---

Scala 2
trait T2U
extends AnyRef

Trait that represents the general interface of the different features needed in order to be able 
to realize a transformation from a type T to a type U

Direct Known Subclasses:
XMI2Scala, PrettyPrint.Scala2Node

<table>
<thead>
<tr>
<th>Value Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
</tr>
<tr>
<td>var</td>
</tr>
<tr>
<td>var</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>abstract</td>
</tr>
<tr>
<td>val</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>def</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>def</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>abstract</td>
</tr>
</tbody>
</table>
| def            | Method that parses an object from a file, initializes gNode, calls back parserHandler 
|               | and makes other operations, if needed. This is basically the only function that should be 
|               | called by a user. |
| protected      | parserHandler : U |
| def            | Method that handles the whole parsing of gNode: it does all the passes needed to 
|               | initialize symbol tables and then creates the associated structures. |

Methods inherited from AnyRef
get, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from Any
==, !=, isInstanceOf, asInstanceOf

Class Summary

abstract class symbTab [tObj] extends AnyRef

Class that contains all relevant information and methods in order to be able to store information about and create objects of type tObj.

Value Details

abstract var gNode : T

The node representing the whole tree to parse.

var otherPass : Boolean

Indicates whether a second parsing of the model will be needed.

abstract val topLev : symbTab[U]

The instance of a concrete implementation of symbTab which is the one used in the parser.

Method Details

protected def parserHandler : U

Method that handles the whole parsing of gNode: it does all the passes needed to initialize symbol tables and then creates the associated structures.

Returns
The created structure

def parseFirstTime : (Int, java.lang.String, Boolean)

Method that realizes the first pass of the symbols preparation.

Returns
the result of prepareSymb(gNode)

def parseAgain : scala.List[Boolean]

Method that realizes the other passes of the symbols preparation.
def createU(x : Int) : U

    Method that creates the different objects.

Parameters
    x - the index in topLev.theTable of the first object to create (it is the index given by parseFirstTime)

Returns
    the created object

abstract def parser(entries : scala.List[T]) : U

    Method that parses an object from a file, initializes gNode, calls back parserHandler and makes other operations, if needed. This is basically the only function that should be called by a user.

Parameters
    files - the list of files needed for all operations of parsing

Returns
    the created object

Scala 2
$\textbf{org.topcased.scaltrans.genericClasses}$

\textbf{trait XMI2Scala}

[source: org/topcased/scaltrans/genericClasses/XMI2Scala.scala]

\begin{verbatim}
trait XMI2Scala[T]
  extends T2U[scala.xml.Node, T]

Trait that represents the interface to create a transformation from an XMI file to Scala objects.

\textbf{Direct Known Subclasses:}
  Ecore2ScalaHelper, Ecore2Scala, Model2Scala
\end{verbatim}

\section*{Value Summary}

\begin{verbatim}
override var gNode : scala.xml.Node

The node representing the whole tree to parse.
\end{verbatim}

\section*{Method Summary}

\begin{verbatim}
def parserFromFile (files : scala.List[java.lang.String]) : T

Method that creates nodes from the files given as parameters and provides a call back to parser with the created nodes.
\end{verbatim}

\section*{Values and Variables inherited from T2U}

\begin{verbatim}
topLev (abstract), otherPass
\end{verbatim}

\section*{Method Summary}

\begin{verbatim}
Methods inherited from T2U

parser (abstract), parserHandler, parseFirstTime, parseAgain, createU
\end{verbatim}

\section*{Methods inherited from AnyRef}

\begin{verbatim}
getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !-, eq, ne, synchronized
\end{verbatim}

\section*{Methods inherited from Any}

\begin{verbatim}
==, !-, isInstanceOf, asInstanceOf
\end{verbatim}

\section*{Value Details}

\begin{verbatim}
override var gNode : scala.xml.Node

The node representing the whole tree to parse.
\end{verbatim}
Overrides
T2U.gNode

Method Details

def parserFromFiles(files : scala.List[java.lang.String]) : T

Method that creates nodes from the files given as parameters and provides a call back to parser with the created nodes.

Parameters
files - the list of the files addresses

Returns
the object corresponding to the parsing of the files
```
object MMScalaHelper extends AnyRef

Object that represents the different structures of the Scala language. But each specific structure will contain relevant information to help to create the corresponding structure using the scala metamodel (see MMScala.scala)

Notes
This is why it is called "helper"

Author
Celia Picard

Version
1.0

Methods inherited from AnyRef
getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from Any
==, !=, isInstanceOf, asInstanceOf

Class Summary

<table>
<thead>
<tr>
<th>case</th>
<th>sAttributeHelper (val name : java.lang.String, val ref : java.lang.String) extends scala.Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Class that represents the helper for the attribute structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case</th>
<th>sClassHelper (val name : java.lang.String, val ref : java.lang.String, val isCaseClass : Boolean, val isObject : Boolean, val attr : scala.List[sAttributeHelper]) extends scala.Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Class that represents the helper for the class structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case</th>
<th>sEnumHelper (val name : java.lang.String, val ref : java.lang.String) extends scala.Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Class that represents the helper for the enum structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case</th>
<th>sOperationHelper (name : java.lang.String, ref : java.lang.String, attr : scala.List[sAttributeHelper]) extends scala.Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Class that represents the helper for the operation structure</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class that represents the helper for the package structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abstract class</th>
<th><code>sStructHelper</code> extends <code>AnyRef</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class that represents any helper structure of the Scala language. All specific helper structures extend this class</td>
</tr>
</tbody>
</table>
object MMScala

Object that represents the Scala Meta-Model. Each class represents a specific structure of the Scala language

Author
Celia Picard

Version
1.0

<table>
<thead>
<tr>
<th>Value Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>val <strong>reservedWords</strong> : scala.Array[java.lang.String]</td>
</tr>
<tr>
<td>Array that contains all the reserved words of the Scala language.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from <strong>AnyRef</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from <strong>Any</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>==, !=, isInstanceOf, asInstanceOf</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>case class <strong>sAbstractCaseClass</strong> (val override name : java.lang.String) extends <strong>sCaseClass</strong></td>
</tr>
<tr>
<td>Class that represents an abstract case class</td>
</tr>
</tbody>
</table>

| case class **sAbstractClass** (val override name : java.lang.String) extends **sClass** with **tAbstractClass** |
| Class that represents an abstract class |

| sealed case class **sAttribute** (val name : java.lang.String, val typ : typable, val iTyp : typeAttr, val ub |
| Class that represents Scala attributes (of a class). An instance of this class will represent an attribute. |

<p>| case class <strong>sCaseClass</strong> (val override name : java.lang.String) extends <strong>sClass</strong> with <strong>tCaseClass</strong> |
| Class that represents a case class |</p>
<table>
<thead>
<tr>
<th>case</th>
<th><code>sCaseObject</code> (val override <code>name : java.lang.String</code>) extends <code>sCaseClass</code> with <code>tObject</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class that represents a case object</td>
</tr>
<tr>
<td></td>
<td>Class that represents a Scala class. An instance of this class will represent a class.</td>
</tr>
<tr>
<td>case</td>
<td><code>sClassAttribute</code> (val override <code>name : java.lang.String</code>, val override <code>typ : tyappable</code>, val override <code>tTyp : typeAttr</code>, val override <code>ub : Int</code>, val override <code>lb : Int</code>, val override <code>constr : Int</code>) extends <code>sAttribute</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents Scala attributes which are instances of other classes</td>
</tr>
<tr>
<td>case</td>
<td><code>sDataAttribute</code> (val override <code>name : java.lang.String</code>, val override <code>typ : tyappable</code>, val override <code>tTyp : typeAttr</code>, val override <code>ub : Int</code>, val override <code>lb : Int</code>, val override <code>constr : Int</code>) extends <code>sAttribute</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents Scala data attributes.</td>
</tr>
<tr>
<td>sealed</td>
<td><code>sDataType</code> (val <code>name : java.lang.String</code>) extends <code>sStruct</code> with <code>tyappable</code> with <code>scala.Product</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents all the different possible data types (Int, String, etc.) Each data type will be represented by a sub-object of this one</td>
</tr>
<tr>
<td>case</td>
<td><code>sEnum</code> (val override <code>name : java.lang.String</code>, val <code>literals : scala.List[(java.lang.String, scala.Option[Int])]) extends </code>sClass`</td>
</tr>
<tr>
<td></td>
<td>Class that represents an enumeration</td>
</tr>
<tr>
<td>case</td>
<td><code>sObject</code> (val override <code>name : java.lang.String</code>) extends <code>sClass</code> with <code>tObject</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents an object</td>
</tr>
<tr>
<td>class</td>
<td><code>sOperation</code> (name : java.lang.String, attr : scala.List[sAttribute], retType : Any, body : Any) extends <code>sStruct</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents Scala operations (methods, functions) of a class</td>
</tr>
<tr>
<td></td>
<td>Class that represents the package structure. An instance of this class will be a specific package.</td>
</tr>
<tr>
<td>case</td>
<td><code>sRefAttribute</code> (val override <code>name : java.lang.String</code>, val override <code>typ : tyappable</code>, val override <code>tTyp : typeAttr</code>, val override <code>ub : Int</code>, val override <code>lb : Int</code>, val override <code>constr : Int</code>) extends <code>sAttribute</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents Scala reference attributes. That means that this attribute refers to another instance of the class typ (which already exists)</td>
</tr>
<tr>
<td>abstract</td>
<td><code>sStruct</code> extends <code>AnyRef</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents any structure of the Scala language. All specific structures extend this class.</td>
</tr>
<tr>
<td>case</td>
<td><code>sTrait</code> (val override <code>name : java.lang.String</code>) extends <code>sClass</code> with <code>tAbstractClass</code></td>
</tr>
<tr>
<td></td>
<td>Class that represents a trait</td>
</tr>
<tr>
<td>trait</td>
<td><code>tAbstractClass</code> extends <code>AnyRef</code></td>
</tr>
<tr>
<td></td>
<td>Trait that indicates that a class is abstract</td>
</tr>
<tr>
<td>trait</td>
<td><code>tCaseClass</code> extends <code>AnyRef</code></td>
</tr>
<tr>
<td></td>
<td>Trait that indicates that a class is a case class</td>
</tr>
<tr>
<td>trait</td>
<td>extends</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>typable</td>
<td>extends</td>
</tr>
<tr>
<td>sealed</td>
<td>typeAttr extends</td>
</tr>
</tbody>
</table>

**Object Summary**

<table>
<thead>
<tr>
<th>case object</th>
<th>extends</th>
<th>dataType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olsList</td>
<td>typeAttr with</td>
<td>scala.Product</td>
<td>Object extending typeAttr and representing the kind of attribute type &quot;List&quot;.</td>
</tr>
<tr>
<td>olsOption</td>
<td>typeAttr with</td>
<td>scala.Product</td>
<td>Object extending typeAttr and representing the kind of attribute type &quot;Option&quot;.</td>
</tr>
<tr>
<td>olsUnique</td>
<td>typeAttr with</td>
<td>scala.Product</td>
<td>Object extending typeAttr and representing the kind of attribute type that are unique (that must exist and are neither a list nor an array).</td>
</tr>
<tr>
<td>sBooleanDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sDoubleDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sFloatDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sIntDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sLongDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sShortDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
<tr>
<td>sStringDataType</td>
<td>extends</td>
<td>sDataType</td>
<td></td>
</tr>
</tbody>
</table>

**Value Details**

```scala
def reservedWords: scala.Array[java.lang.String]
val reservedWords : scala.Array[java.lang.String] =
```

Array that contains all the reserved words of the Scala language.
# org.topcased.scaltrans.mmparser

## object Tools

This trait contains utility methods used in particular by MMParser and MMParserHelper.

### Author
Celia Picard

### Version
1.0

## Method Summary

<table>
<thead>
<tr>
<th>def</th>
<th><strong>createClass</strong> (name : java.lang.String, supClass : java.lang.String, body : java.lang.String) : java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method that pretty prints the class declaration scala code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method that creates the scala code corresponding to a method composed of a match/case instruction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method that creates the scala code corresponding to a method composed of a match/case instruction with the matching object being an instance of a class (in opposition with match over a string)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method that creates the scala code corresponding to a method composed of a match/case instruction with the matching object being a String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method that creates the scala code corresponding to a method composed of a match/case instruction with the matching object being an object and that gives a name (name) to the object of each option.</td>
</tr>
</tbody>
</table>
def createImport (listImports : scala.List[java.lang.String], sp : java.lang.String) :
  java.lang.String
Method that pretty prints import instructions in scala code

def createInnerMatch (err : java.lang.String, matchingObj : java.lang.String, rightPartArrow :
  (sClass) => java.lang.String, sp : java.lang.String, pack : java.lang.String, classes :
scala.List[sClass]) : java.lang.String
Method that creates the scala code corresponding to a match/case instruction with the
matching object being an object

def makeListAttr (latr : scala.List[sAttribute], rightPartArrow : (sAttribute) =>
Method that pretty prints attributes under a list form

Methods inherited from AnyRef
getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq,
ne, synchronized

Methods inherited from Any
==, !=, isInstanceOf, asInstanceOf

Method Details

def createFunctionMatchOverObject(err : java.lang.String, header : java.lang.String, matchingObj :
java.lang.String, rightPartArrow : (sClass) => java.lang.String, sp : java.lang.String, pack :
java.lang.String, classes : scala.List[sClass]) : java.lang.String

Method that creates the scala code corresponding to a method composed of a match/case
instruction with the matching object being an instance of a class (in opposition with match over
a string)

Parameters
  err  -  the header of the error case (package_name.class_name.function_name)
header  -  the header of the method (name, params, etc.)
matchingObj  -  the name of the object on which the match is done
rightPartArrow  -  method that indicates in each case the right part of a case arrow
sp  -  the indentation spaces to be put before the header
pack  -  the package the classes belong to
classes  -  the different case classes of the match

Returns
  The string corresponding to the match method

def createFunctionMatchOverString(err : java.lang.String, header : java.lang.String, matchingObj :
java.lang.String, rightPartArrow : (sClass) => java.lang.String, sp : java.lang.String, classes :
scala.List[sClass]) : java.lang.String
Method that creates the scala code corresponding to a method composed of a match/case
instruction with the matching object being a String
Parameters
err - the header of the error case (package_name.class_name.function_name)
header - the header of the method (name, params, etc.)
matchingObj - the name of the object on which the match is done
rightPartArrow - method that indicates in each case the right part of a case arrow
sp - the indentation spaces to be put before the header
classes - the different case classes whose name will correspond to the cases of the match

Returns
The string corresponding to the match method


Method that creates the scala code corresponding to a method composed of a match/case instruction with the matching object being an object and that gives a name (name) to the object of each option.

Parameters
err - the header of the error case (package_name.class_name.function_name)
header - the header of the method (name, params, etc.)
matchingObj - the name of the object on which the match is done
name - the name that will be given to the cases options
rightPartArrow - method that indicates in each case the right part of a case arrow
sp - the indentation spaces to be put before the header
pack - the package the classes belong to
classes - the different case classes whose name will correspond to the cases of the match

Returns
The string corresponding to the match method


Method that creates the scala code corresponding to a method composed of a match/case instruction

Parameters
err - the header of the error case (package_name.class_name.function_name)
header - the header of the method (name, params, etc.)
matchingObj - the name of the object on which the match is done
sp - the indentation spaces to be put before the header
cases - a list composed of (left part of the arrow (matching case), right part of the arrow)

Returns
The string corresponding to the match method

Method that creates the scala code corresponding to a match/case instruction with the matching object being an object

**Parameters**

- err - the header of the error case (package\_name.class\_name.function\_name)
- matchingObj - the name of the object on which the match is done
- rightPartArrow - method that indicates in each case the right part of a case arrow
- sp - the indentation spaces that is put before the header
- pack - the package the classes belong to
- classes - the different case classes whose name will correspond to the cases of the match

**Returns**

The string corresponding to the match method

```python
def makeListAttr(lattr : scala.List[sAttribute], rightPartArrow : (sAttribute) => java.lang.String, comp : java.lang.String) : java.lang.String
```

Method that pretty prints attributes under a list form

**Parameters**

- lattr - the list of attributes to be pretty printed
- rightPartArrow - method that indicates the way to print each attribute
- comp - the composition operator to be put between the different composants of the list (basically, it can we ":=" or "::") to the cases of the match

**Returns**

The string corresponding to the pretty printed list of attributes

**Examples**

```python
makeListAttr(a :: b :: List(), rpa, "::") returns rpa(a) :: rpa(b) :: List()
makeListAttr(a :: b :: List(), rpa, "::") returns rpa(a) :::: rpa(b)
```

```python
```

Method that pretty prints import instructions in scala code

**Parameters**

- listImports - the list of the paths of the classes we want to import
- sp - the indentation spaces to be put before the import

**Returns**

"sp import import1 \n sp import import2 \n sp import import3 \n ..."

```python
```

Method that pretty prints the class declaration scala code

**Parameters**

- name - the name of the class
- supClass - the name of the super class (if no super class : ")"
body - the body of the class (methods, functions, variables, etc.)

Returns
"class name [extends supClass] \{\nbody \n\}"
org.topcased.scaltrans.mmparser

object HelperFileCreator

[source: org/topcased/scaltrans/mmparser/HelperFileCreator.scala]

object HelperFileCreator extends AnyRef

Object that offers the methods to create the helper file from the meta model file.

Author
Celia Picard

Version
1.0

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>def createHelperFile (sourceFile : java.lang.String, destFile : java.lang.String) : Unit</td>
<td>Method that creates the helper file from the metamodel file. This is basically the only function that should be used by a user.</td>
</tr>
<tr>
<td>private ppAttribute (o : scala.xml.Node, path : java.lang.String) : java.lang.String</td>
<td>def Method that pretty prints the attribute contained in node</td>
</tr>
<tr>
<td>private ppClass (o : scala.xml.Node, path : java.lang.String) : java.lang.String</td>
<td>def Method that pretty prints the classifier contained in node</td>
</tr>
<tr>
<td>private prettyPrint (o : scala.xml.Node) : java.lang.String</td>
<td>def Method that pretty prints the helper file</td>
</tr>
<tr>
<td>private prettyPrintThis (path : java.lang.String)(o : scala.xml.Node) : java.lang.String</td>
<td>def Method that pretty prints the node</td>
</tr>
</tbody>
</table>

Methods inherited from AnyRef

getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from Any

==, !=, isInstanceOf, asInstanceOf

Method Details

Method that creates the helper file from the metamodel file. This is basically the only function that should be used by a user.

Parameters
  sourceFile - address of the file containing the metamodel
  destFile - address where to store the helper file

private def prettyPrint(o: scala.xml.Node): java.lang.String

Method that pretty prints the helper file

Parameters
  o - the node containing the metamodel

Returns
  The corresponding string.


Method that pretty prints the node o

Parameters
  path - the path of the node we want to pretty print
  o - the node we want to pretty print

Returns
  The corresponding string


Method that pretty prints the package contained in node o

Parameters
  o - the node containing the package we want to pretty print
  path - the path of the node we want to pretty print

Returns
  The corresponding string


Method that pretty prints the classifier contained in node o

Parameters
  o - the node containing the classifier we want to pretty print
  path - the path of the node we want to pretty print

Returns
  The corresponding string

Method that pretty prints the enumeration contained in node o

Parameters
  o – the node containing the enumeration we want to pretty print
  path – the path of the node we want to pretty print

Returns
  The corresponding string


Method that pretty prints the attribute contained in node o

Parameters
  o – the node containing the attribute we want to pretty print
  path – the path of the node we want to pretty print

Returns
  The corresponding string

Scala 2
org.topcased.scaltrans.mmparser

class Ecore2ScalaHelper

[source: org/topcased/scaltrans/mmparser/Ecore2ScalaHelper.scala]

class Ecore2ScalaHelper
extends XMI2Scala[sStructHelper]

Class that allows to parse a meta-model helper (from a xml file), to create the corresponding
structures in the Scala helper meta-model (MMScalaHelper). The structures created will then
be used when parsing the meta-model to create the appropriate structures in the Scala
meta-model (sStruct).

Author
Celia Picard

Version
1.0

Value Summary

<table>
<thead>
<tr>
<th>var</th>
<th>creTabAttributes : scala.List[sAttributeHelper]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The list of attributes created during parsing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
<th>creTabClasses : scala.List[sClassHelper]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The list of classes created during parsing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
<th>creTabPackages : scala.List[sPackageHelper]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The package created during parsing</td>
</tr>
</tbody>
</table>

override topLev : obj
val symbTab : obj
The instance of a concrete implementation of symbTab which is the one used in the
parser.

Values and Variables inherited from XMI2Scala

gNode

Values and Variables inherited from T2U

otherPass

Method Summary

override def parser (files : scala.List[scala.xml.Node]) : sStructHelper
Method that parses an object from a file, initializes gNode, calls back parserHandler
and makes other operations, if needed. This is basically the only function that should be
called by a user.
Methods inherited from `XMI2Scala`

- `parserFromFile`

Methods inherited from `T2U`

- `parserHandler`, `parseFirstTime`, `parseAgain`, `createU`

Methods inherited from `AnyRef`

- `getClass`, `hashCode`, `equals`, `clone`, `toString`, `notify`, `notifyAll`, `wait`, `wait`, `wait`, `finalize`, `==`, `!=`, `eq`, `ne`, `synchronized`

Methods inherited from `Any`

- `==`, `!=`, `isInstanceOf`, `asInstanceOf`

**Object Summary**

<table>
<thead>
<tr>
<th>private object</th>
<th>attrHelperTab extends symbTab[sAttributeHelper]</th>
</tr>
</thead>
<tbody>
<tr>
<td>private object</td>
<td>classHelperTab extends symbTab[sClassHelper]</td>
</tr>
<tr>
<td>object</td>
<td>obj extends symbTab[sStructHelper]</td>
</tr>
<tr>
<td></td>
<td>Object that realizes the interface between the different types of objects we can find in an Ecore metamodel helper (EClass, EStructuralFeatures, ...) and allows to make the parsing of a metamodel helper providing calls back to the appropriate functions for each feature.</td>
</tr>
<tr>
<td>private object</td>
<td>packageHelperTab extends symbTab[sPackageHelper]</td>
</tr>
</tbody>
</table>

**Value Details**

Var `creTabClasses` : `scala.List[sClassHelper]`

The list of classes created during parsing

Var `creTabAttributes` : `scala.List[sAttributeHelper]`

The list of attributes created during parsing

Var `creTabPackages` : `scala.List[sPackageHelper]`

The package created during parsing

Override val `topLv` : `obj`

The instance of a concrete implementation of `symbTab` which is the one used in the parser.
Method Details

override def parser(files : scala.List[scala.xml.Node]) : sStructHelper

    Method that parses an object from a file, initializes gNode, calls back parserHandler and makes other operations, if needed. This is basically the only function that should be called by a user.

Parameters
    files - the list of files needed for all operations of parsing

Returns
    the created object
org.topcased.scaltrans.mmparser

class Ecore2Scala

[source: org/topcased/scaltrans/mmparser/Ecore2Scala.scala]

class Ecore2Scala extends XMI2Scala[sStruct]

Class that allows to parse a meta-model (from a xml file), to create the corresponding structures in the Scala meta-model (MMScalea), to pretty print this structure in Scala code (and write it to a file) and to create the files that allow to parse, transform and pretty print models corresponding to this meta model.

Author
Celia Picard

Version
1.0

Value Summary

<table>
<thead>
<tr>
<th>private</th>
<th>allClassesCreated : scala.List[sClass]</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>The list of all classes created during parsing (useful for the creation of transformer)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>private</th>
<th>argsClasses : scala.List[sClass]</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>The list of all the classes that are types of an argument (useful for the creation of transformer)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
<th>classHelper : scala.List[sClassHelper]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The list of classes created when parsing the helper meta-model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>private</th>
<th>concreteClassesCreated : scala.List[sClass]</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>The list of concrete classes (not abstract, not trait) created during parsing (useful for the creation of parser, transformer and pretty-printer)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>private</th>
<th>gSupClass : java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>The super class of all the classes of the meta-model (by default, it is Any)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
<th>packageName : java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The name of the package we are creating. This will be set automatically by createSPackage. And will be used to create the files XXXParser, XXXPrettyPrint and XXXTransfo.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>var</th>
<th>prefixPackage : java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The name of the package in which the one we are going to create is embedded. If there is no, its value is &quot;&quot;.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>private</th>
<th>referencedClasses : scala.List[sClass]</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>The list of all the classes that are types of reference arguments (useful for the creation of transformer)</td>
</tr>
</tbody>
</table>
override val topLev: obj

The instance of a concrete implementation of symbTab which is the one used in the parser.

Values and Variables inherited from XMI2Scala

gNode

Values and Variables inherited from T2U

otherPass

Method Summary

private def createEDataType (ind: Int): sStruct

def createFiles (files: (java.lang.String, java.lang.String, java.lang.String)): Unit

Method that creates the code corresponding to XXPParser, XXPrettyPrint and XXTransfo and prints them to the corresponding files.

private def createXXParser: java.lang.String

Method that creates the text corresponding to the scala code of the model parser, for models corresponding to this meta-model.

private def createXXPrettyPrint: java.lang.String

Method that creates the text corresponding to the scala code of the model pretty printer, for models corresponding to this meta-model.

private def createXXTransfo: java.lang.String

Method that creates the text corresponding to the scala code of the model transformer, for models corresponding to this meta-model.

def.ecore2Scala (files: scala.List(java.lang.String)): Unit

Method that realizes the whole transformation from an Ecore metamodel to the printing of the files to use in order to parse models (according to the meta-model)

private def getDataType (name: java.lang.String): sDataType

Method that creates a sDataType from a string containing the name of the corresponding data type in ecore syntax

private def getPrefix: java.lang.String

Method that gets the prefix of the meta-model

private def getURI: java.lang.String

Method that gets the URI of the meta-model

override def parser (files: scala.List[scala.xml.Node]): sStruct

Method that parses an object from a file, initializes gNode, calls back parserHandler and makes other operations, if needed. This is basically the only function that should be called by a user.

def pn: java.lang.String

Method that returns the complete name of the parser, depending on the value of prefixPackage
Methods inherited from **XML2Scala**

- parserFromFile

Methods inherited from **T2U**

- parserHandler, parseFirstTime, parseAgain, createU

Methods inherited from **AnyRef**

- getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from **Any**

- ==, !=, isInstanceOf, asInstanceOf

---

**Object Summary**

<table>
<thead>
<tr>
<th>private object</th>
<th>attrTab extends symbTab[sAttribute]</th>
</tr>
</thead>
<tbody>
<tr>
<td>private object</td>
<td>classTab extends symbTab[sClass]</td>
</tr>
<tr>
<td>private object</td>
<td>enumTab extends symbTab[sEnum]</td>
</tr>
<tr>
<td>object</td>
<td>obj extends symbTab[sStruct]</td>
</tr>
</tbody>
</table>

Object that realizes the interface between the different types of objects we can find in an Ecore metamodel (EClass, EStructuralFeatures, ...) and allows to make the parsing of a metamodel providing calls back to the appropriate functions for each feature.

| private object | packTab extends symbTab[sPackage] |

---

**Value Details**

**private var** concreteClassesCreated : scala.List[sClass]

The list of concrete classes (not abstract, not trait) created during parsing (useful for the creation of parser, transformer and pretty-printer)

**private var** allClassesCreated : scala.List[sClass]

The list of all classes created during parsing (useful for the creation of transformer)

**private var** referencedClasses : scala.List[sClass]

The list of all the classes that are types of reference arguments (useful for the creation of transformer)
private var argsClasses : scala.List[sClass]

The list of all the classes that are types of an argument (useful for the creation of transformer)

var classHelper : scala.List[sClassHelper]

The list of classes created when parsing the helper meta-model

private var gSupClass : java.lang.String

The super class of all the classes of the meta-model (by default, it is Any)

var packageName : java.lang.String

The name of the package we are creating. This will be set automatically by createSPackage. And will be used to create the files XXParser, XXPrettyPrint and XXTransfo.

var prefixPackage : java.lang.String

The name of the package in which the one we are going to create is embedded. If there is no, its value is "".

override val topLev : obj

The instance of a concrete implementation of symbTab which is the one used in the parser.

## Method Details

private def getPrefix : java.lang.String

Method that gets the prefix of the meta-model

**Returns**

gNode.attribute("nsprefix") .get.toString

private def getURI : java.lang.String

Method that gets the URI of the meta-model

**Returns**

gNode.attribute("nsURI") .get.toString

private def createEDataType(ind : Int) : sStruct

private def getDataType(name : java.lang.String) : sDataType
Method that creates a sDataType from a string containing the name of the corresponding data type in.ecore syntax

**Parameters**
- name - the name in.ecore syntax of the data type we want to create

**Returns**
The corresponding sDataType

override def parser(files : scala.List[scala.xml.Node]) : sStruct

Method that parses an object from a file, initializes gNode, calls back parserHandler and makes other operations, if needed. This is basically the only function that should be called by a user.

**Parameters**
- files - the list of files needed for all operations of parsing

**Returns**
the created object

def ecore2Scala(files : scala.List[java.lang.String]) : Unit

Method that realizes the whole transformation from an Ecore metamodell to the printing of the files to use in order to parse models (according to the meta-model)

**Parameters**
- files - the list of files to use. The files must be given in the following order:
  - files(0): the helper file
  - files(1): the file containing the metamodel
  - files(2): the file where to print the scala code corresponding to the metamodel
  - files(3): the file where to print the class XXPParser
  - files(4): the file where to print the class XXPrettyPrint
  - files(5): the file where to print the class XXTransf

def createFiles(files : (java.lang.String, java.lang.String, java.lang.String)) : Unit

Method that creates the code corresponding to XXPParser, XXPrettyPrint and XXTransf and prints them to the corresponding files.

**Parameters**
- files - a tuple with the names of the files:
  - _1: the file where to print XXPParser
  - _2: the file where to print XXPrettyPrint
  - _3: the file where to print XXTransf

def pn : java.lang.String

Method that returns the complete name of the parser, depending on the value of prefixPackage
private def createXXParser : java.lang.String

Method that creates the text corresponding to the scala code of the model parser, for models corresponding to this meta-model.

This text is meant to be printed to a file in order to be compiled and used afterwards.

**Returns**
The corresponding text

---

private def createXXPrettyPrint : java.lang.String

Method that creates the text corresponding to the scala code of the model pretty printer, for models corresponding to this meta-model.

This text is meant to be printed to a file in order to be compiled and used afterwards.

**Returns**
The corresponding text

---

private def createXXTransfo : java.lang.String

Method that creates the text corresponding to the scala code of the model transformer, for models corresponding to this meta-model.

This text is meant to be printed to a file in order to be compiled and used afterwards.

**Returns**
The corresponding text

---

Scala 2
org.topcased.scaltrans.mmparser
object WholeMM2Scala

[source: org/topcased/scaltrans/mmparser/WholeMM2Scala.scala]

object WholeMM2Scala extends AnyRef

Objects that allows to parse a metamodel composed of various files (or not) Basically, if this object is used, then the user should only call methods of this object as it provides all the appropriate call back to the methods of other classes.

<table>
<thead>
<tr>
<th>Value Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>val hm : scala.collection.mutable.HashMap[Java.lang.String, Ecore2Scala]</td>
</tr>
<tr>
<td>HashMap that makes links between a String (name of the package) and an instance of Ecore2Scala.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function that realizes the whole parsing process of a metamodel.</td>
</tr>
</tbody>
</table>

Methods inherited from AnyRef
getClass, hashCode, equals, clone, toString, notify, notifyAll, wait, wait, wait, finalize, ==, !=, eq, ne, synchronized

Methods inherited from Any
==, !=, isInstanceOf, asInstanceOf

<table>
<thead>
<tr>
<th>Value Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>val hm : scala.collection.mutable.HashMap[Java.lang.String, Ecore2Scala]</td>
</tr>
<tr>
<td>HashMap that makes links between a String (name of the package) and an instance of Ecore2Scala.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function that realizes the whole parsing process of a metamodel.</td>
</tr>
</tbody>
</table>
Function that realizes the whole parsing process of a metamodel.

**Parameters**

- **packName** - the name of the global package in which the package contained in each file is embedded
- **mm** - a list of tuples composed as follow, for each package of the metamodel:
  - _1_: address of the metamodel helper file
  - _2_: address of the metamodel file
  - _3_: address of the file where to write the Scala code corresponding to the metamodel
- **files** - a list of tuples composed as follow, for each package of the metamodel:
  - _1_: address of the file where to write the Scala code corresponding to XXParser
  - _2_: address of the file where to write the Scala code corresponding to XXPrettyPrint
  - _3_: address of the file where to write the Scala code corresponding to XXTransfo

---

**Scala 2**
### org.topcased.scaltrans.parser

**class Model2Scala**

[source: org/topcased/scaltrans/parser/Model2Scala.scala]

abstract class Model2Scala[T]
   extends XMI2Scala[T]

   Class that offers the general methods that allow to parse a model according to a meta-model. To create a parser for a specific meta-model this class must be extended and some specific methods for the meta-model must be written. The corresponding class can be generated automatically thanks to the createXXParser method of the class MMParser.

**Parameters**

T - the super type of all the classes of the meta-model. If there is no, put Any

**Author**

Celia Picard

**Version**

1.0

<table>
<thead>
<tr>
<th>Value Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>protected abstract val</strong></td>
</tr>
<tr>
<td>The complete name of the package that contains the parser</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values and Variables inherited from XMI2Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>gNode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Values and Variables inherited from T2U</th>
</tr>
</thead>
<tbody>
<tr>
<td>otherPass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract method to be implemented in a concrete parser class.</td>
</tr>
</tbody>
</table>

<p>| How to get attributes of a node. |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scala.List((java.lang.String, scala.xml.Node))</code></td>
<td>Method that returns the list of all the attributes of the node o that correspond to the class name in the meta-model.</td>
</tr>
<tr>
<td><code>getAttr (o : scala.xml.Node, attr : scala.List[tAttr]) : scala.List((java.lang.String, scala.xml.Node))</code></td>
<td>Method that returns the list of all the data attributes of the node o.</td>
</tr>
<tr>
<td><code>getAttrOC (o : scala.xml.Node, attr : scala.List[tAttr]) : scala.List((java.lang.String, scala.xml.Node))</code></td>
<td>Method that returns the list of all the class attributes of the node o.</td>
</tr>
<tr>
<td><code>getAttrRef (o : scala.xml.Node, attr : scala.List[tAttr]) : scala.List((java.lang.String, java.lang.String))</code></td>
<td>Method that returns the list of all the reference attributes of the node o.</td>
</tr>
<tr>
<td><code>listAttr (name : java.lang.String) : scala.List[tAttr]</code></td>
<td>Abstract method to be implemented in a concrete parser class.</td>
</tr>
<tr>
<td><code>parser (files : scala.List[scala.xml.Node]) : T</code></td>
<td>Method that parses a model according to a meta-model and generates the appropriate structures. This is basically the only method that should be used by a user.</td>
</tr>
<tr>
<td><code>verify (e : T) : Boolean</code></td>
<td>Abstract method to be implemented in a concrete parser class.</td>
</tr>
</tbody>
</table>

Methods inherited from **XM12Scala**

- `parserFromFile`

Methods inherited from **T2U**

- `parserHandler`, `parseFirstTime`, `parseAgain`, `createU`

Methods inherited from **AnyRef**

- `getClass`, `hashCode`, `equals`, `clone`, `toString`, `notify`, `notifyAll`, `wait`, `wait`, `wait`, `finalize`, `==`, `!=`, `eq`, `ne`, `synchronized`

Methods inherited from **Any**

- `==`, `!=`, `isInstanceOf`, `asInstanceOf`

**Class Summary**

```scala
```

The class that represents all the information needed for attributes.

**Object Summary**
object **TTab** extends symbTab[T]

Object that contains all relevant methods to handle the parsing of attributes.

### Value Details

**override val topLev : TTab**

The instance of a concrete implementation of symbTab which is the one used in the parser.

**protected abstract val packageName : java.lang.String**

The complete name of the package that contains the parser.

### Method Details


Abstract method to be implemented in a concrete parser class.

This method must return a method that takes as arguments:

- The list of data attributes of the class `name`. The type of this argument is: `List[List[Any]]`

  Each data attribute must be included in a list: `(List(a1)) :: a2 :: ... :: List()`, where `a2` is of type `List[X]`

- The list of class attributes of the class `name`. The type of this argument is: `List[List[T]]`

  Each class attribute must be included in a list: `(List(a1)) :: a2 :: ... :: List()`, where `a2` is of type `List[X]`

- The list of reference attributes of the class `name`. The type of this argument is: `List[List[T]]`

  Each reference attribute must be included in a list: `(List(a1)) :: a2 :: ... :: List()`, where `a2` is of type `List[X]`

  and that creates an instance of the class `name` of the meta-model

**Parameters**

- `name` - the name of the meta-model class we want to create an instance of.

**Returns**

The corresponding method

**protected abstract def listAttr(name : java.lang.String) : scala.List[tAttr]**

Abstract method to be implemented in a concrete parser class.
This method must return the list of attributes of the class name of the meta-model.

**Parameters**

name - the name of the meta-model class we want the attributes of.

**Returns**

A list of tuples of tAttr

---

```
protected abstract def verify(e : T) : Boolean
```

Abstract method to be implemented in a concrete parser class.

This method must return a boolean representing whether the attributes of the class name are correct according to the meta model (size, etc.)

**Parameters**

e - the part of the expression of type τ we want to check.

**Returns**

The corresponding boolean.

---

```
```

Method that returns the list of all the attributes of the node o that correspond to the class name in the meta-model.

**Parameters**

o - the node containing the structure we want the attributes of
name - the name of the corresponding structure in the meta-model

**Returns**

A tuple of lists (one list for each kind of attributes). It is constructed as follow:
- _1_: the list of data attributes (result of getAttr)
- _2_: the list of reference attributes (result of getAttrRef)
- _3_: the list of class attribute (result of getAttrOC)

---

```
```

Method that returns the list of all the class attributes of the node o.

**Parameters**

o - the node containing the structure we want the class attributes of
attr - the list of attributes for the type of node o.

**Returns**

A list of tuples constructed as follow: (name of the attribute, the attribute).

---

```
```


Method that returns the list of all the data attributes of the node o.

Parameters
o - the node containing the structure we want the data attributes of
attr - the list of attributes for the type of node o.

Returns
A list of tuples constructed as follow : (name of the attribute, the attribute).


Method that returns the list of all the reference attributes of the node o.

Parameters
o - the node containing the structure we want the reference attributes of
attr - the list of attributes for the type of node o.

Returns
A list of tuples constructed as follow : (name of the attribute, the attribute).

override def parser(files : scala.List[scala.xml.Node]) : T

Method that parses a model according to a meta-model and generates the appropriate structures. This is basically the only method that should be used by a user

Parameters
files - address of the file containing the model

Returns
The corresponding structure

Scala 2
class PrettyPrint

abstract class PrettyPrint[T]
   extends AnyRef

   Class that offers the general functions that allow to pretty print, in an XML syntax, a model
   according to a meta-model. To create a pretty printer for a specific meta-model this class must
   be extended and some specific functions for the meta-model must be written. The
   corresponding class can be generated automatically thanks to the createXXPrettyPrint
   function of the class MMParse.

   Parameters
   T   – the super type of all the classes of the meta-model. If there is no, put Any

   Author
   Celia Picard

   Version
   1.0

Value Summary

| abstract  | prefix : java.lang.String |
|           | Corresponds to the prefix of the meta-model. Must be initialized when extending this class. |

| abstract  | xmlns : java.lang.String |
|           | Corresponds to the URI of the meta-model. Must be initialized when extending this class. |

Method Summary

| abstract  | getNameClass (e : T) : java.lang.String |
|           | Abstract method to be implemented in a concrete prettyPrint class. |

| abstract  | listAttr (e : T) : scala.List[(java.lang.String, java.lang.String)] |
|           | Abstract method to be implemented in a concrete prettyPrint class. |

|           | Abstract method to be implemented in a concrete prettyPrint class. |

| abstract  | listAttrRef (e : T) : scala.List[(java.lang.String, java.lang.String, Boolean)] |
|           | Abstract method to be implemented in a concrete prettyPrint class. |

| def       | prettyPrint (exp : T) : java.lang.String |
|           | Method that realizes the complete pretty printing of an expression (transforms it to
|           | nodes and the pretty prints it). This is basically the only method that should be called by |
a user.

Methods inherited from `AnyRef`  
`getClass`, `hashCode`, `equals`, `clone`, `toString`, `notify`, `notifyAll`, `wait`, `wait`, `wait`, `finalize`, `==`, `!=`, `eq`, `ne`, `synchronized`

Methods inherited from `Any`  
`==`, `!=`, `isInstanceOf`, `asInstanceOf`

Object Summary

```scala
object Scala2Node extends T2U[T, scala.xml.Node]
Object that implements all the functionalities needed to convert Scala expression to Nodes.
```

Value Details

```scala
abstract val `prefix` : java.lang.String  
```
Corresponds to the prefix of the meta-model. Must be initialized when extending this class.

```scala
abstract val `xmlns` : java.lang.String  
```
Corresponds to the URI of the meta-model. Must be initialized when extending this class.

Method Details

```scala
```
Abstract method to be implemented in a concrete prettyPrint class.

This method must return the list of class attributes of `e` in the meta-model. For each class attribute, it must call the method `createListAttrOC`.

**Parameters**

- `e` - the expression we want the class attributes of.

**Returns**

The list of the result of `createListAttrOC` for each class attribute of the class of `e`

```scala
abstract def `listAttr`(e : T) : scala.List[(java.lang.String, java.lang.String)]
```
Abstract method to be implemented in a concrete prettyPrint class.

This method must return the list of data attributes of `e` in the meta-model. For each class, it must return a list of tuples : (name of attribute, value of attribute)
Parameters
   e - the expression we want the class attributes of.

Returns
   A list of (name of attribute, value of attribute)

abstract def listAttrRef(e : T) : scala.List[(java.lang.String, java.lang.String, Boolean)]

   Abstract method to be implemented in a concrete prettyPrint class.

   This method must return the list of reference attributes of e in the meta-model. For each
   reference attribute, it must call the method lookUpRef.

Parameters
   e - the expression we want the class attributes of.

Returns
   The list of the result of lookUpRef for each class attribute of the class of e

abstract def getNameClass(e : T) : java.lang.String

   Abstract method to be implemented in a concrete prettyPrint class.

   This method returns the name of the class of e

Parameters
   e - the expression we want the name of the class.

Returns
   The name of the class of e

def prettyPrint(exp : T) : java.lang.String

   Method that realizes the complete pretty printing of an expression (transforms it to nodes and
   the pretty prints it). This is basically the only method that should be called by a user.

Parameters
   exp - the expression we want to parse

Returns
   the corresponding String (xmi format)