Université de Franche-Comté
Supervisers : Fabien Peureux, Isabelle Jacques

# Java Generation from UML Models specified with Alf Annotations

Supervised project report

Alexandre Vernotte
Jean-Marie Gauthier
Master of computer science
2010-2011

# Acknowledgments

First of all, we are deeply indebted the project initiatiors, Mr. Fabien Peureux and Mrs. Isabelle Jacques from LIFC, for their assistance and valuable advices about the plug-in development.

We also wish to thank Mr. Bruno Tatibouet from LIFC who had the kindness of receiving us to make light on various meta-modeling technics.

Furthermore, we would like to give our special thank to the Smartesting team, particularly to Mrs. Severine Colin who has given us time's work to show us differents technics about Eclipse project and UML model recovering.

Finally, thanks to Mrs. Marie Rivolet and Mr. Jonathan Lambert, for their precious advices concerning redaction and oral project's presentation.

# Abstract

The model driven engineering (MDE [1]) proposes to place models in heart of the development. Actually, it's not necessary to develop a program with several thousand lines of code because recents modelers, such as Papyrus, offers modules to generate code from UML models. The plug-in for Papyrus developed here is in this mind of putting models at the center of development. It allows Java code generation from UML diagram and ALF specification. The plug-in does not cover the whole expressive power of UML and ALF. However, it is a good introduction to further progress in this direction : increase of the ALF grammar, meta-model more complete, etc.

# Table des matières

# Introduction

This document is about the creation of a plug-in in the modeling environment Papyrus. This software allows the creation of UML models that can represent different types of system (software, physical systems, etc). It is then possible to use models to generate code, for example. This is what engineering model driven proposes to do : the source code is not modified but the model is.

Models have always been present in many industrial fields but the habit is to baptizing them "plans". By analogy, a software model is a kind of map software conceived before beginning its development. Moreover, the main asset of software models is that it is possible to automate part of their development, verification and validation. So they bring a considerable increase of safety and time saving.

Since August 2010, the OMG [2] has raised its range of normalized languages with the action language ALF (Action Foundational Language for UML). Thus, it becomes possible to define operations's behaviors in a model using a language close to those used for development : indeed, the ALF's language syntax is deliberately very close to the Java programming language. This OMG's choice tends to enhance the benefits of modeling because it is now possible to include, in a static structure (class diagram), the dynamic aspects of the modeled system.

So, this project for papyrus offers to achieve an Eclipse plug-in, allowing generation of Java code from a UML class diagram, whose operations are explained using the ALF language. Therefore, this document sets out the context and objectives of the development of a plug-in for Papyrus. In addition, it presents the stages of development and implementation of this tool.

# Chapitre 1

# Objectives and context

This chapter is about presenting on the one hand the “Top-Down” design which has been followed, and on the other hand the technologies and tools used to create the plug-in, like the eclipse platform ,the ALF language and Papyrus.

## 1.1 Top-Down approach

To fulfill the project, it was necessary to follow a good strategy. The “Top-down” approach was chosen : it’s “a design of information processing and knowledge ordering, mostly involving software”. Basically, it starts by formulating an overview of a system, and refining it into subsystems. Then those subsystems are specified and can be refined again into smaller subsystems, until there are only base elements left.

In this case, it first started by creating a use case to specify the system, then this use case helped establishing the user needs, which helped defining the features required in the plug-in.

### 1.1.1 Use Case

To get a proper idea of what needed to be created, a scenario has been written, starring a self-employed man called Jojo. He has a specific request, but is also having money trouble and therefore will have to find a “lowcost” solution, the alf2java plug-in. Here it is :

Jojo needs a software but can't afford to hire a company to do the job. He has some knowledge about UML modelling though, so he decides to go for it and starts the conception using PAPYRUS. He eventually gets what looks like a viable model, and then decides to generate Java sources from his diagram. He takes a look at the code, but can't see through it : he just doesn't get Java. But then, Jojo remembers he took a course back in the days about imperative programming. Maybe he could use that, he doesn't really want to go all over Java... After some research on the internet, he finds the solution : a plug-in for PAPYRUS that allow users to specify class operations with ALF code, and translate the whole thing into Java. Jojo looks up on the manual to see if he understands the ALF syntax, and finds it very close to what he knows. He can start filling the blank.

On every operation Jojo simply goes on the "properties" chapter and add a body dedicated to ALF (see Fig. 4.1).

Now Jojo can specify the operation behaviour (Fig. 4.2).

*(Another solution would be to open a text editor in a new tab, allowing edition, syntax highlighting and on-the-fly code parsing, it would enhance ALF conception but is definitely harder to do).*

When every operation has been specified, Jojo wants to translate everything into Java. A button on the PAPYRUS interface triggers this particular task, he then clicks on it. The program tells him he forgot to mention what language he wants to generate, so he goes on the configuration chapter and chooses Java (as it's the only way to keep the ALF). Jojo restarts the generation, but there are new errors now, about ALF this time. One of them says he forgot a semicolon at the end of an instruction, another one says a class variable on the code doesn't exist on the model. He therefore corrects the model and runs the generation for the third time. It's a glowing success ! Everything went fine, all the classes have their Java equivalent, their operations have been filled with executable code, the project is runnable. Jojo finally saves his work and exits the application.

### 1.1.2 User Needs

Considering the situation above, it's easy to define what users will need when using the plug-in. First of all, they will need to specify every class methods with ALF code. Then, to know if they did everything right, they will want to check their work, check if there is no incompatibility between the model and what has been specified. After, they will want to generate runnable Java code from the model, as it is the main goal of the MDE approach. Finally, they will have to save their work, and be able to open it later.
Let's now find out which features should be included in the program.

### 1.1.3 Features

The next step of the "Top-Down" method is deducting the modules that will be created, from the user needs. Here is what will be developed :

- **ALF code parser** : To check if the user wrote understandable ALF code.
- **Dedicated metamodel** : it will help gather information about diagrams without having to use UML2 API's, which are complex.
- **Types checker** : To check if the user did not forgot to declare variables, and did respect their scope.
- **ALF to java translater** : its name speaks for itself.
- **Java Generator** : it will give executable Java code from a metamodel instance.
- **Core** : it will hold all the other modules together. It will call them in the right order to perform their task.
- **Buttons and menus in the UI** : It's not a module but it is not less important. Indeed, users need trigger button to launch the plug-in.

## 1.2 Technologies and tools

In this section are presented the technologies and tools used in this project. To begin with, the Eclipse platform and one of its plug-in, Papyrus, and then the ALF language.

### 1.2.1 Eclipse and Papyrus

Eclipse is an open-source multi-languages development platform written in Java, comprising an IDE (Integrated Development Environment) and an extensible plug-in system. It is used to create applications in many different languages, such as C, C++, Java, PHP, ADA, Perl, Python, etc... All of Eclipse's functionalities are plug-ins, even its runtime system, plugged together following a dependencies hierarchy. Everybody can contribute to the project by creating its own plug-in.

Papyrus is an open-source modeling tool and an Eclipse plug-in created in 2003 by CEA (Commissariat à l'énergie atomique et aux énergies alternatives). Not very popular at first, it is now coming to maturity and is rising up as a very efficient modeler. Because it is a plug-in, it needs existing technologies such as UML2, EMF, GMF, etc... In fact, it uses their functionnalities to achieve some of its tasks.

### 1.2.2 ALF language

ALF (Action Language for Foundational UML) is an action language recently created to model UML class diagrams programmatically. ALF specifications were provided on August 2010 by OMG (Object Management Group), a consorsium created in 1989, focused on modeling (programs, systems and business processes) and model-based standards. OMG has set standards for at least one hundred and fifty langages, like UML, XML, C, Cobol, Corba, ADA, and many others. With this language, a model can be completely designed by code, although this plug-in will not use the entire ALF grammar : the objective is to use both graphic and programmatical technics. Users will design classical UML models, with the ability to specify each class method with a subdivision of the ALF grammar.

# Chapitre 2

# Development

This part focus on the development of all the modules. It can be divided in three stages : Firstly, a lot of effort was put into data recovery. Secondly, those datas had to be processed. And thirdly, when everything was recovered, checked, processed, the last task to perform was the Java Generation.

## 2.1 Data recovery

Because Eclipse is a very complex entity, getting the right information from it is far from being easy. A huge set of methods exists, and sometimes even the internet could not help. The first data to recover was the active project the user is working on. In eclipse, there is not such method as *"getActive-Project()"*. And even from a random project, it was very hard to extract the UML model, because it is load into memory and therefore there is no obvious method to retrieve it either.

The development was frozen, so the group had a two hours technic meeting with one of Smartesting's engineer, Severine Colin [3]. She helped by showing the methodology to adopt for accessing various datas, by getting the current selected element first, which is much easier to do. This meeting was really useful because it unfroze the project, as either the active project and its associated model could now be retrieved.

## 2.2 Data processing

Once information has been gathered, it will have to be checked to prevent errors from happening. This section concentrates on the data processing modules : the ALF parser, the Types checker, and the dedicated metamodel.

### 2.2.1 ALF parser

As ALF is a full language, it has its own grammar. A module was developed to make sure this grammar is respected, by deploying lexical and syntactic analyzers. A lexical analyzer focus on decomposing a text into a sequence of tokens (for example, words), sequence that will be consumed by a syntactic analyzer, to determine its grammatical structure and see if it matches a specific formal grammar. To create those analyzers, the JavaCC [**?**] application was used : it is an open-source parser generator for Java.

Every ALF specification get to be tested, and if one of the tests comes negative, an error will be thrown.

### 2.2.2 Types Checker

Knowing that ALF code respects its grammar is not enough to generate error-free Java code. Another important thing to check is whether the variables in the annotation really exist when they are called. In fact, there are two types of variable, class attributes and local variables. Class attributes are declared graphically, with papyrus, and local variables are declared programmatically in ALF annotations. There are then two types of verifications : everytime the parser hits a variable call, it has to check if this variable has been declared and is still in its declaration scope, and if the test fails it has to check if this variable is a class attribute. If both tests fail, the current parsing throws an error.

### 2.2.3 Dedicated Metamodel

A diagram represents a model for a specific kind of application. A metamodel could be described as a model for models : it is also represented by a diagram and can hold a specific kind of model, in this case UML models. Creating a dedicated metamodel means several things. First, it is because the existing UML2 metamodel is way to complicated and a none negligeable part of it was useless to the plug-in. And then, it is also because by having a independant metamodel it is possible to implement specific methods to it, for instance methods for java generation.

The Alf metamodel is composed of six elements : APackage, AClass, AAttribute, AOperation, AParameter, AAnnotation. Each one of them has methods to return its attributes (for example, its instance's name), and a method to convert itself into Java.

## 2.3 Java generation

The last remaining operation before getting the plug-in's final product, i.e. runnable java code, is the generation itself. There are actually two jobs : translating ALF annotation into Java, and generating packages and Java classes from the metamodel.

### 2.3.1 ALF to Java Translater

Even if ALF and Java have a very similar syntax, a translation is still necessary. For that, a design pattern called *"visitor"* was implemented. In fact, JavaCC creates it after successively parsing Annotations. It is a programmatical representation of a syntactic tree, composed of nodes. Each node refers to a grammar rule, and has an associated *"visit()"* method, which means it has a specific behaviour.

### 2.3.2 Java Generator

It was said earlier, every element composing the metamodel has a method to convert itself into Java. These methods are very simple : they are simply returning their element's Java structure in a String. The first thing to do is creating Java files according to the user's diagram. Then, to create a complete Java Class, the module calls AClass generation method, which calls the other element's methods. The result is a String, it is injected into a file previously created which name matches a class of the diagram.

### 2.3.3 Core

The core is kind of a scheduler : it calls all the other modules in the right order. It also handle errors, stopping the plug-in execution and displaying their message into the console. Moreover, every contribution to the UI (User interface) was done in the core by using the “extension points”, another tool from eclipse that facilitate contributions to plug-ins by others, without changing the existing code.

# Chapitre 3

# Conclusion

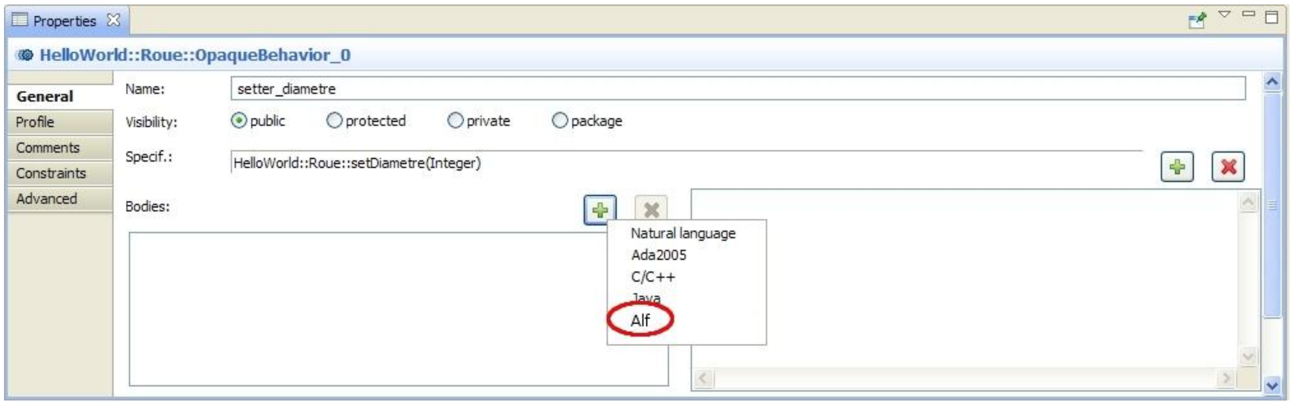
This plug-in, written in Java, can specifies UML operations of class diagram with ALF code. It also checks it and finally, translates it into Java code. The code generation is based on a meta-model developed specifically for the project. It provides a framework for a subset of UML and ALF that the plugin can treats. The meta-model can also checks the consistency between the UML model and ALF annotations. Creating a parser and a dedicating meta-model allows the plug-in to avoid all the external constraints that may influence his future evolution. The plug-in is totally independant.

Finally, the expressive power of UML and ALF that the plugin can accepts, is still minimal : it only accepts two primitives types (Integer and Boolean), a variable declaration statement, an if statement and finally an assignment statement of a class variable by using the "this" keyword. However, the plug-in is functional and provides efficient evolution instead of the implementation of the whole expressive power.

# Chapitre 4

# Annexes

## 4.1 Figures


HelloWorld::Roue::OpaqueBehavior_0

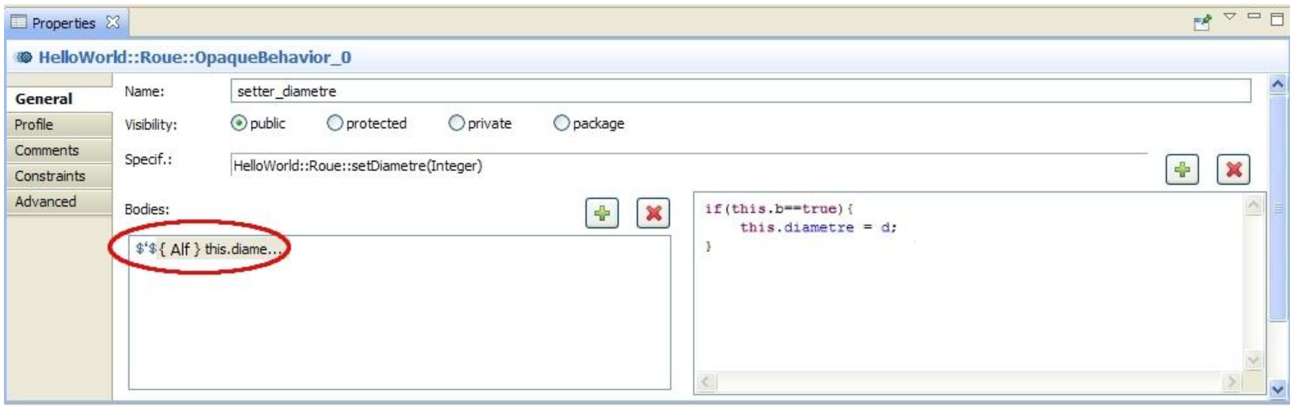
Name: setter_diametre

Visibility: [✓ public] protected private package

Specif.: HelloWorld::Roue::setDiametre(Integer)

Bodies:

Fig. 4.1 – Add ALF body


Properties
HelloWorld::Roue::OpaqueBehavior_0

General | Profile | Comments | Constraints | Advanced

Name: setter_diametre
Visibility: [✓ public] protected private package
Specif.: HelloWorld::Roue::setDiametre(Integer)
Bodies: $‘$ { Alf } this.diame...

```
if(this.b==true){
    this.diametre = d;
}
```

Fig. 4.2 – Specify operation behaviour

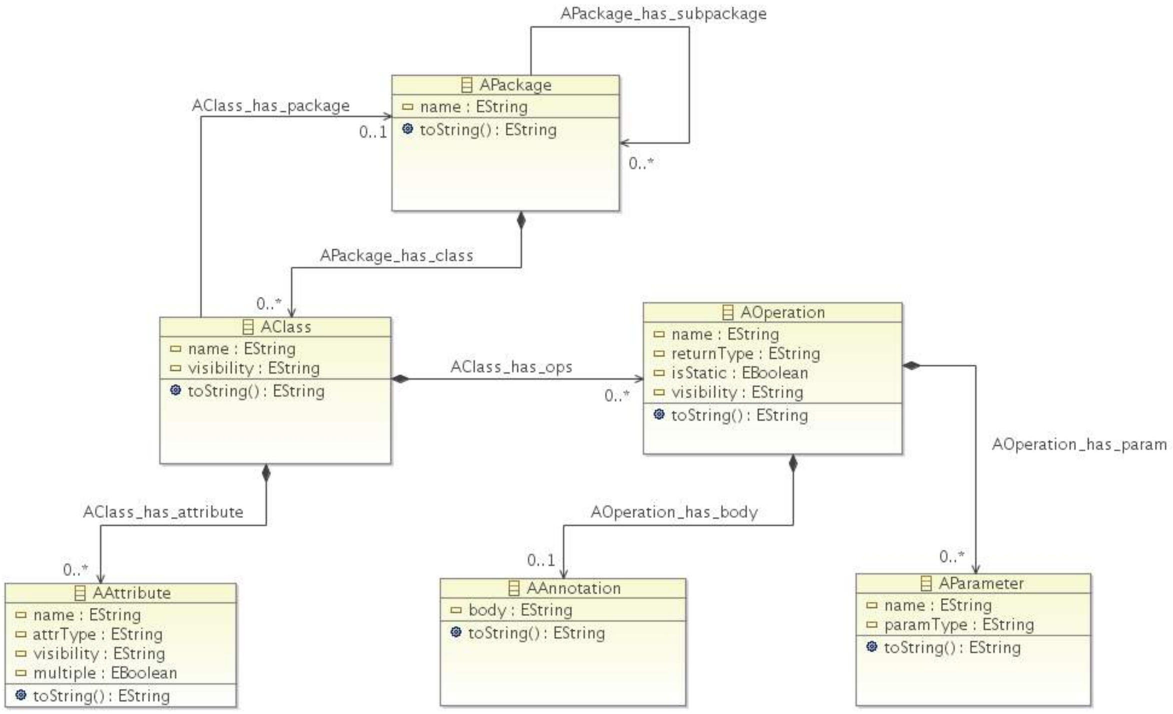


Fig. 4.3 – Our Meta Model Diagram

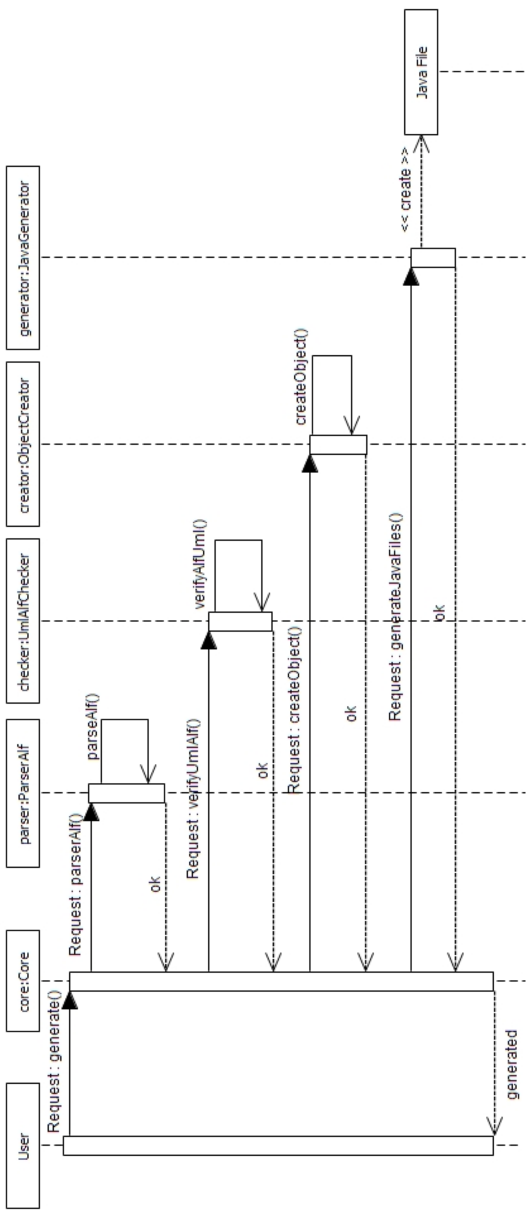


Fig. 4.4 – Sequence Diagram

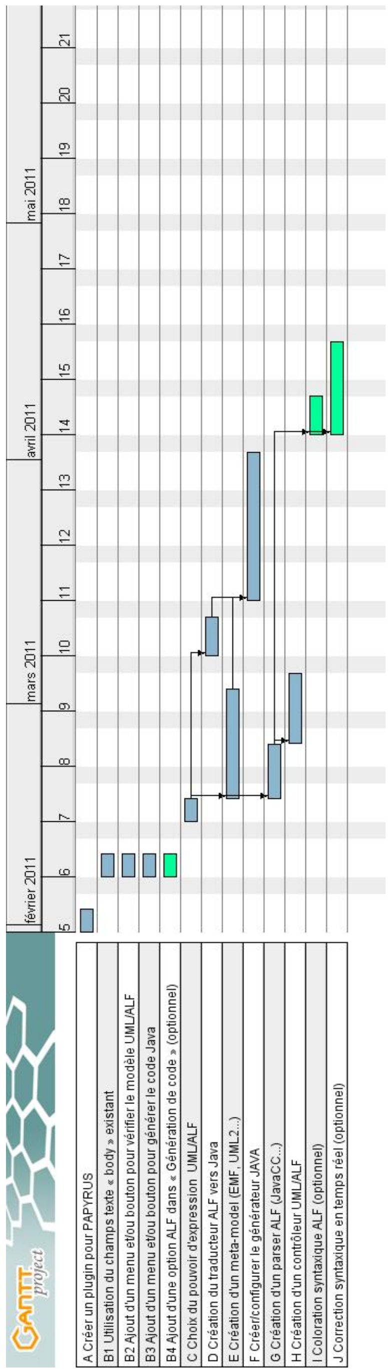


Fig. 4.5 – Provisional Schedulel

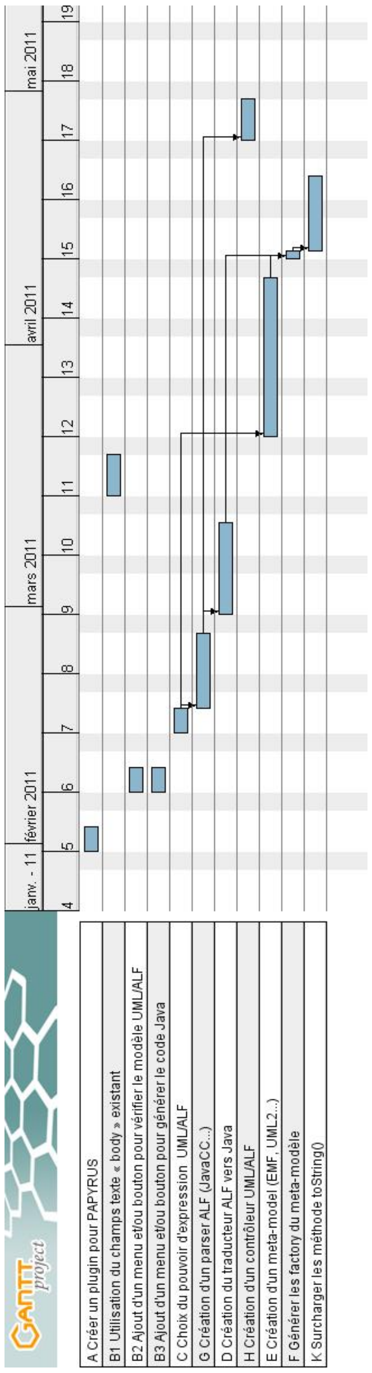


Fig. 4.6 – Real Schedule

# Bibliographie

[1] Usine Logicielle. Idm, 2010. http://www.usine-logicielle.org/index.php?option=com_content&task=view&id=44&Itemid=28.

[2] OMG. Alf specification, Aug 2010. http://www.omg.org/spec/ALF/.

[3] Smartesting. Smartesting, 2008. http://www.smartesting.com/.