UNIVERSITY OF GOTHENBURG

A DSL Supporting Textual and Graphical Views

Master of Science Thesis in Software Engineering

SALOME MARO

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, March 2014

A DSL Supporting Textual and Graphical Views

SALOME MARO

Examiner: MIROSLAW STARON
Supervisor: MATTHIAS TICHY

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2014.

Abstract

Domain Specific Languages(DSLs) are languages that are designed to be used in a particular development area. These languages aim to help developers solve the problems related to that domain and therefore contain information and jargons that are only relevant to a particular domain. Domain specific languages can be expressed in textual or graphical formats. Apart from personal preferences there are several advantages of using graphical format and also several advantages of using textual format. Therefore having a DSL that supports both of these notations will mean harvesting the advantages of all of them. However most of the tools available that enable the use of domain specific languages tend to focus on either textual or graphical editors for the DSL. The aim of this thesis is to investigate the possibility of having both notations for the same DSL in use. The thesis was conducted using action research method at Ericsson AB. Ericsson is currently having a DSL that has only a graphical notation. This DSL is using UML and UML Profiles. A prototype of the textual version of the existing DSL was created using Xtext and used to make an analysis and come up with findings on how a DSL with both graphical (which is in UML) and textual notations can be used. Transformations that enable switching from one view of the model to another have also been prototyped and used for analysis. The thesis also investigated two other solutions that are based on EMF using Xtext for text and GMF for the graphical notation. This thesis concludes that with all the alternatives investigated, it is possible to have a DSL that supports both graphical and textual views. Each solution however varies in the effort needed to implement and maintain the DSL.

Acknowledgements

I would like to extend my sincere gratitude to my industry supervisor Lars Gelin and the entire CBB team at Ericsson Kista for the constant support and help that they offered during the Thesis work. I would also like to thank my university supervisor Matthias Tichy for the support and continuous input that he gave for this thesis. Lastly I would like to thank my family for the encouragement they provided throughout the thesis.

Salome Maro, July 2014

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1
- 1.1 Background 1
- 1.2 Problem Statement 2
- 1.3 Purpose 2
- 1.4 Disposition of the Report 3

2 Foundations 4
- 2.1 Model Driven Engineering 4
- 2.2 UML 5
- 2.3 EMF 6
- 2.4 Xtext 6
- 2.5 GMF 8
- 2.6 Model Transformations 9
- 2.7 The Ericsson DSL 10

3 Method 13
- 3.1 Research Questions 15
- 3.2 Hypotheses 15

4 Related Work 17

5 Solution 19
- 5.1 Alternative 1 19
  - 5.1.1 UML Profile to a Metamodel in Ecore 20
  - 5.1.2 Generating the Textual Editor using Xtext 23
  - 5.1.3 Switching between Graphical and Textual Views 24
  - 5.1.4 Case 1 31
  - 5.1.5 Case 2 41

5.1.6 Results and Analysis . . . 43
5.1.7 Discussion . . . 48
5.2 Alternative 2 . . . 49
5.2.1 Prototype . . . 51
5.2.2 Discussion . . . 51
5.3 Alternative 3 . . . 52
5.3.1 Discussion . . . 54

6 Threats to Validity 55
6.1 Construct Validity . . . 55
6.2 Internal Validity . . . 55
6.3 External Validity . . . 56
6.4 Reliability Validity . . . 56

7 Conclusions and Future Work 57
7.1 Summary . . . 57
7.2 Conclusions . . . 57
7.3 Future Work . . . 60

A Higher Order Transformation to Generate a transformation from UML to Hive text instance model 62

B ATL Transformation to Transform a UML Profile into an Ecore model 73

Bibliography 79

List of Figures

2.1 UML Diagrams overview[1]. 5
2.2 Xtext Editor. 8
2.3 GMF Dashboard. 9
2.4 Model to model transformation. 10
2.5 The Hive Profile composition. 11
2.6 Code generation steps (From a Hive model to .c and .h files). 12

3.1 Action research cycle [2]. 14
3.2 Cycles of Action research implemented. 16

5.1 Alternative One. 20
5.2 UML Stereotypes to Ecore classes. 22
5.3 ATL Higher Order Transformation Chain[3]. 28
5.4 Simplified Version of the ATL metamodel[4]. 29
5.5 Simplified Version of the ATL trace model [4]. 30
5.6 Sample trace model. 31
5.7 Part of the Hive Behavior Profile, its UML subset and the resulting Ecore after transformation. 32
5.8 Model created with Xtext editor. 33
5.9 Obtaining an Ecore model for use in Xtext. 43
5.10 Errors due to missing names in the model. 46
5.11 Alternative two. 50
5.12 GMF and Xtext editors Integrated. 51
5.13 Alternative three. 53

List of Tables

5.1 UML to Ecore Mapping 23
5.2 Demo model contents 45

7.1 A summary of the three alternatives investigated 58

Chapter 1

Introduction

1.1 Background

The term Domain Specific Language (DSL) refers to a language that is created for the purpose of being used in a specific domain. They are different from General Purpose Languages (GPL) as their creation is done in order to serve specific needs of some domain and not general needs. A DSL captures design patterns that are common in a particular domain making it easy for developers to create models using these design patterns. A model in this case is a simplification or an abstraction of a system built with a specific purpose in mind [5]. Compared to General Purpose Languages, a DSL focuses on the jargons and patterns that are used in a specific domain therefore avoiding all the general notations that are not needed within the domain. This makes creation of applications much easier since the languages are tailored to fit their particular domain and designers can write much less code (as little as 2%) than when using a GPL [6]. Applications written using DSLs tend to be more concise, easier to maintain and reason about and above all can quickly be written [7]. This is due to the fact that DSL offers an abstraction that is higher compared to using GPLs. Models created with DSLs can be subjected to code generation techniques to generate code in any chosen programming language.

Companies adopting the use of Domain Specific Languages (DSL) usually have various reasons to do so. For Ericsson the adoption of DSLs is mainly to raise the abstraction with which engineers create applications. This gives the engineers an opportunity to be able to focus on the problem at hand and not worry about the implementation details. The fact that software at Ericsson is usually deployed on various hardware platforms is another main reason for the adoption of DSLs. It would be very hard and inefficient for engineers to write different code for every hardware platform, but with DSLs they can reuse the logic models and only change the deployment model for the different hardware platforms. The logic models together with the deployment models are used to generate code that is suitable for different hardware platforms. This proves to be more time efficient and reduces the probability of having errors in the code. A DSL can have a textual notation, graphical notation and sometimes even tabular notation.

1.2 Problem Statement

When the DSL is small, having only one notation can be a feasible thing to do, but when a DSL is large, covers a wider aspect and has different types of users having one notation may not suit the needs for these users. This is because some use cases are easier to specify when using graphical notations while others can be conveniently specified using textual notations [8].

Using graphical notations for creating applications has advantages like reducing the chances of errors, providing visualization and hence ease understanding of the system being created. However using text based modeling also has advantages like speed of creation and editing, speed of formatting and wide availability of editors [8]. It would be best if all these advantages can be harnessed. So a solution that will support the use of both graphical and textual notation in an easy and effective way is of great importance.

Currently, Ericsson has only the graphical notation of the DSL and as stated previously some designers would prefer to use text when modeling and in some cases, the use cases can better be understood when modeled using text. So the first problem is the lack of a textual version of the DSL. However, having a DSL with both the textual and graphical version being used simultaneously raises several more general problems to be addressed. One of the main problems is how to maintain the two DSLs without adding a lot of maintenance effort to the company. This means that in case the DSLs needs to be updated one should not have to do a lot of manual work. There is a need for a solution that will provide some automation to make it easy to update the language.

Another problem that arises is how the end users can be able to switch between the graphical and textual views in an easy way and without losing any information. The idea here is that a user should be able to only press a button and switch between the views and can edit the models in any of the views.

1.3 Purpose

The purpose of this thesis is to investigate the possibilities of having a DSL with both graphical and textual representation. The thesis also investigates the possibility of switching between textual and graphical concrete syntax without any loss of information. The work done in this thesis contributes to the previous research that has been done on having several notations for a DSL by providing prototypes that show how textual and graphical notations can be used for one DSL in one development environment.

What has been done throughout the thesis is the creation of a prototype of the textual version of the existing DSL at Ericsson using Xtext and used this prototype to make an analysis and come up with findings on how a DSL with both graphical (which is in UML) and textual notations can be used. Prototypes of transformations that enable switching from one view of the model to another have also been implemented and used for analysis. The thesis also investigates two other solutions that are based on EMF using Xtext for textual notation and GMF for the graphical notation. This thesis draws the conclusion that it is possible to have a DSL that supports both graphical and textual views after investigating the three alternative solutions. Each solution however varies in the effort needed to implement and maintain the DSL. The thesis also provides insight on how syntax errors are handled when switching from one view to another using an inconsistent model.

1.4 Disposition of the Report

The remainder of this thesis report is organized as follows. Chapter 2 gives a brief introduction of the technologies that have been used during the thesis and chapter 3 describes the research method that was used to conduct the thesis. Chapter 4 provides a summary of the related work. Chapter 5 describes the solution to the problem and three alternatives that have been investigated during the research. For each alternative its general concept is first described and then its application on a case/prototype is described. This section also provides findings from each alternative in the discussion subsections. Chapter 6 discusses the threats to validity associated with this thesis. Lastly, the thesis ends with Chapter 7 which provides a summary of what has been done, conclusions drawn, answers to research questions and proposals for future work.

Chapter 2

Foundations

2.1 Model Driven Engineering

Model driven engineering is a term referring to a phenomena where systems or applications are created using models. Models as mentioned earlier are a simplification or abstraction of a system created with a specific purpose in mind [5]. With model driven engineering, designers create an abstraction/ model of the system that is needed. These models are independent of the actual platform that will later create the running applications. This means that, for the same model running applications can be generated in C, Visual basic, Java or any other programming language using code generation techniques.

In order to create models, designers need a language. The languages in model driven engineering are divided into two main types. The first type is known as General Purpose Language (GPL) , these are languages that are fit for use in almost any domain. They are called general purpose because they fit in a number of domains. UML is a good example of a GPL. The second type is known as Domain Specific Language (DSL). These are languages defined for use in a particular domain. They are usually small and concise with information only needed for the domain addressed. SQL is an example of a Domain Specific Language made for the database domain.

Defining a DSL requires two steps which are abstract syntax definition and concrete syntax definition. The first step which is abstract syntax definition is where the modeling concepts and their properties are defined. Here we create a metamodel that defines all the allowed (valid) models of the modeling language [9]. The second step which is the concrete syntax definition is where we define the notation of the language. The concrete syntax defines what language elements (graphical or textual) are associated with which metamodel element. It is where we link metamodel elements with a visual representation

[10].This notation can either be textual or graphical notation and it is also possible to define different notations for the same language. Graphical DSLs use pictorial symbols to represent the application being modeled while Textual DSLs use text to represent the application being modeled. The definition of both the abstract syntax and concrete syntax for a DSL needs tools that can facilitate this. There are several frameworks that are designed to facilitate creation and use of DSLs. However the Eclipse Modeling Framework (EMF) is the most prominent in the world [9]. Throughout this thesis EMF is used as the modeling framework. EMF is described in details in chapter 2.3.

2.2 UML

UML stands for Unified Modeling Language, it is one of the most used modeling languages defined by the Object Management Group(OMG). The language uses graphics to represent models. UML is a general purpose language meaning that it can be used in any domain and to model any type of application. UML provides 13 types of diagrams that can be used for modeling [11]. These diagrams are categorized into three main categories. The first category is the Structure category that consists of Class Diagram, Package Diagram, Object Diagram, Composite Structure Diagram, Component Diagram and Deployment Diagram. The second category is Behavior which consists of Activity diagrams, State Machine diagrams and Sequence diagrams. The third category is Interaction and it consists of Timing diagram, communication diagram, interaction overview diagram and also the sequence diagram [11].

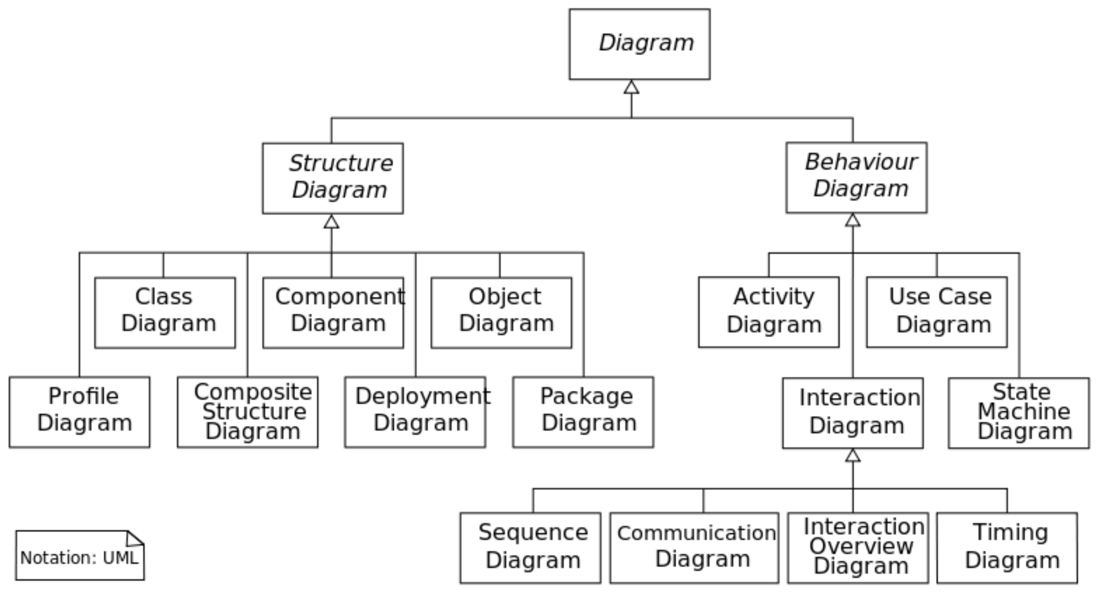


FIGURE 2.1: UML Diagrams overview[1].

Even though UML is a GPL, it provides a mechanism for it to be extended to include features fitting a particular domain. This mechanism for extension is called UML Profile and allows for customization of syntax and semantics of UML [12]. With UML Profiles one can define extra features that are not offered by UML out of the box and use the profile when modelling with UML. UML Profiles enable the definition of additional classes which are known as Stereotypes, additional attributes which are known as tagged values. A stereotype extends a metaclass which is a class that already exist in UML. When a stereotype is applied to an instance of a metaclass, the class will have the extra attributes and references that is added by the stereotype. In other words the resulting class will have attributes and references of the metaclass as well as attributes and references of the stereotype class. When using UML with profiles, the language moves from the GPL side toward the DSL side as the profile provides information that is specific to a particular domain [13].

2.3 EMF

EMF stands for Eclipse Modeling Framework. As the name suggests, it is a framework that is used to facilitate creation and manipulation of models using the Eclipse Integrated Development Environment (IDE). With EMF installed on Eclipse one is able to create models and edit models. EMF also comes with code generation functionality for creating tools and other applications based on models [14]. One of the core functionality of EMF is that it enables the definition of metamodels using a metamodel called Ecore [9]. Ecore is an EMF metamodel [15] and other models can be created conforming to it. When a model is created conforming to the Ecore metamodel it is saved in an Ecore format(.Ecore). Due to the prominence of EMF in the model driven engineering world, many other tools and plugins in model driven engineering are created based on it. Examples of these tools are GMF[16], Xtext[17], ATL[18] and EMFText [19]. There are also several UML tools built on EMF for instance Papyrus which is an open source tool and RSA-RTE which is a proprietary tool from IBM.

2.4 Xtext

As mentioned previously, DSLs can be implemented with a textual or graphical concrete syntax. There are several tools that can be used to create Textual concrete syntax but for this thesis XText has been used. XText is a plugin that is built on Eclipse modeling framework (EMF) and can be used to create programming languages and textual DSLs [17]. Xtext has been selected because the current tooling used with the existing graphical

DSL at Ericsson is also based on EMF. Xtext also provides a lot of functionality out of the box. Some of the functionality provided are Syntax highlighting, outline overview, scoping, model reference validation, quick fixes and contest assist [17].

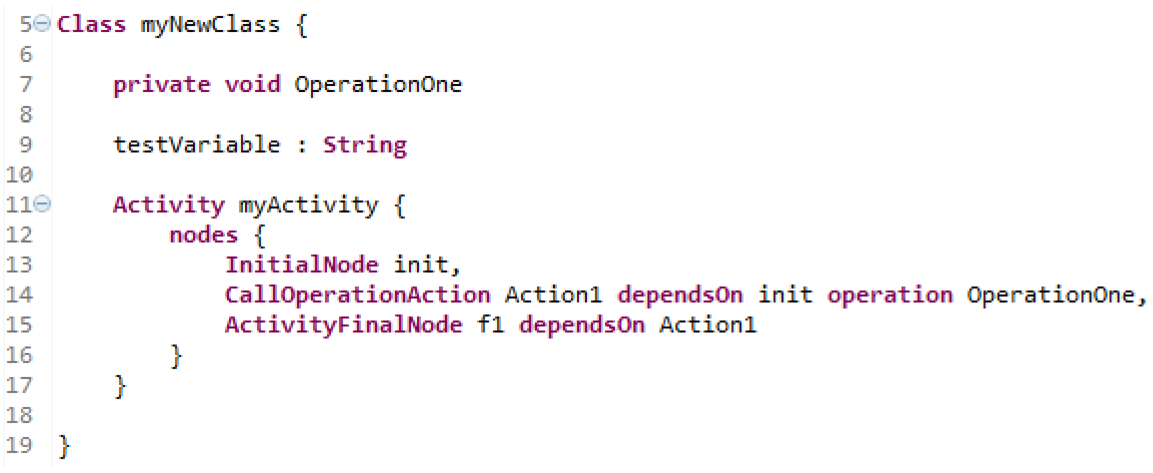
All languages need a grammar. A grammar is a set of structural rules for a language. It determines what kind of word combinations (phrases) are valid in the language. For a textual DSL as well, a grammar needs to be defined. In Xtext, a language grammar is defined using rules. These rules determine how this model element will be represented in text. It is in the grammar where keywords of a DSL are also defined. An example of a grammar rule for a model element called Class is given below:

```
 1 Class_Impl returns Class:
 2   {Class}
 3   'Class'
 4   name=EString
 5   '{'
 6     (ownedOperation+=Operation ( "," ownedOperation+=Operation)* )?
 7     (ownedAttribute+=Property ( "," ownedAttribute+=Property)* )?
 8     (ownedConnector+=Connector ( "," ownedConnector+=Connector)*  )?
 9     (ownedPort+=Port ( "," ownedPort+=Port)*  )?
10     (ownedBehavior+=Behavior ( "," ownedBehavior+=Behavior)*  )?
11     (extension_HiveBaseBehaviorClass=HiveBaseBehaviorClass)?
12     (extension_HiveStructureClass=HiveStructureClass)?
13     (extension_HiveInstanceRouterClass=HiveInstanceRouterClass)?
14     (extension_Capsule=CapsuleStereotype)?
15   '}';
```

Listing 2.1: Xtext rule for a Class

The rule name is Class_Impl, which in this case is short for Class Implementation. The word ‘Class’ in blue color indicates a keyword. This rule means that if one wants to define a class it has to start with a keyword ‘Class’, followed by name of the class, followed by an opening curly brace. After the opening curly brace one can define the attributes and references belonging to the class (represented by line 6 to 15). The ? symbol means that the attribute or reference is optional. According to the above rule a closing curly brace is needed to end the definition of a class.

There are two approaches that can be used to create textual DSLs when it comes to XText. The first approach is when there is no metamodel that exists for the DSL. This way one starts to define the grammar of the language and the concrete syntax, and then XText will automatically generate a metamodel for the language. The second approach is when a metamodel for the DSL already exists. In this approach, the metamodel is given as input to XText and XText generates the grammar and concrete syntax based on the given metamodel. The generated grammar and concrete syntax may not suit what the user wants, for instance it contains curly braces and keywords for every rule and this may be unnecessary in the grammar. However this generated grammar is not permanent, it can be further edited to fit the needs of the users. When one has defined the grammar, Xtext has the ability to generate the parser and an editor that can be used for the language defined. The figure below shows an example of an editor created with Xtext. The editor displays a class defined by the user according to the rule in figure 2.1 above.



```
Class myNewClass {

    private void OperationOne

    testVariable : String

    Activity myActivity {
        nodes {
            InitialNode init,
            CallOperationAction Action1 dependsOn init operation OperationOne,
            ActivityFinalNode f1 dependsOn Action1
        }
    }

}
```

FIGURE 2.2: Xtext Editor.

2.5 GMF

GMF stands for Graphical Modeling Framework. It is an EMF based framework for creating graphical DSL editors [16]. With GMF, you can define what symbols and shapes can be used to represent elements in your DSL. Definition of a diagram editor using GMF relies on a metamodel for the DSL which should be defined using Ecore. This metamodel is also known as domain model. Using this domain model, GMF can generate the graphical definition model and tooling definition model. The graphical definition model defines the nodes and connections in association with the domain model (metamodel of the DSL). The nodes are the shapes for instance rectangles, rounded rectangles, ellipse e.t.c. The connections are usually arrows that connect shapes to each other. The tooling definition model defines the tools to be displayed in the editor's menu. These are the tools that will be used to add nodes and connections to the editor. GMF also generates a mapping model which maps each graphical definition which is a node or connection to a specific tool defined in the tooling definition model. For example if in your domain model you have a class called Package and you want this Package class to be represented with a rectangle shape in your editor you would need to define it as follows. In the graphical definition model you would select a shape that is a rectangle for your package, in the tool definition model you would define a tool and name it Package and in the mapping model you would map the tool to the shape defined in the graphical definition model. The mapping model is used to generate the diagram editor gen-model which is then used to generate the diagram editor for the DSL. The figure below shows the GMF dashboard. The dashboard shows the models and how they relate to each other.

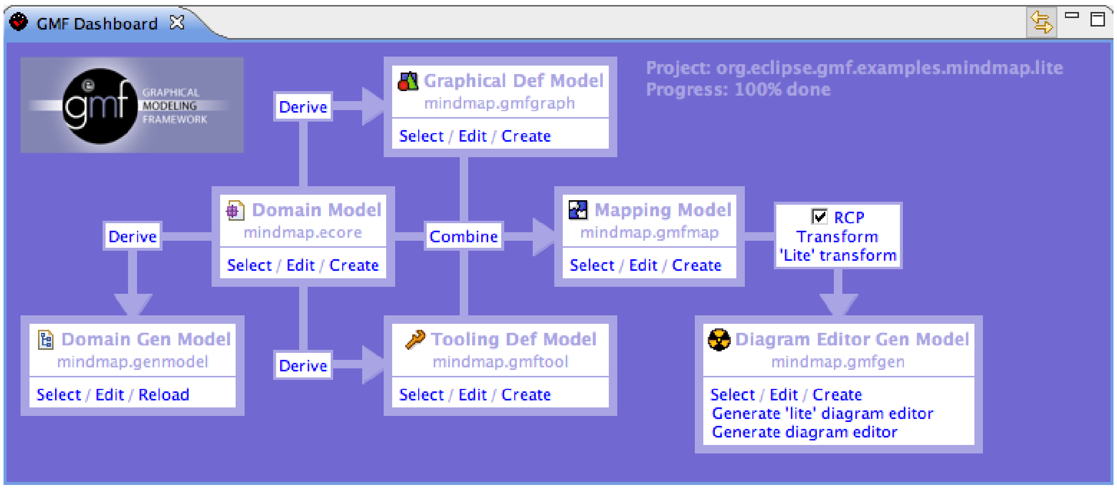


FIGURE 2.3: GMF Dashboard.

2.6 Model Transformations

A model conforming to one metamodel can also be transformed to a model conforming to another metamodel. A model transformation is usually defined on a metamodel level where the relationship between one metamodel to another (transformation rules) is defined. This transformation is then used to transform instance models conforming to the metamodels. A transformation needs one or more input model(s) (source models) and can produce one or more output model(s) (target models).

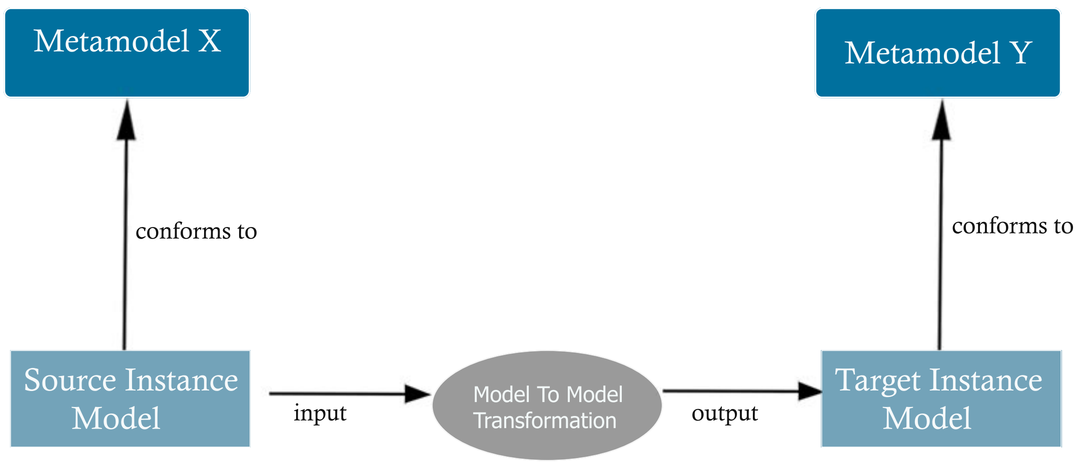


Figure 2.4: Model to model transformation.

The figure above illustrates a model to model transformation where models conforming to metamodel X are transformed to models conforming to metamodel Y. The transformation takes an instance model conforming to metamodel X as a source and produces an instance model conforming to Y as a target. Model transformations can either be written manually by a programmer or can be automatically generated using some Higher Order Transformations (HOT). The generation of transformations is possible since a transformation is also a model that conforms to some transformation metamodel [3]. For instance a transformation written in ATL is indeed a model that conforms to the ATL metamodel.

Currently there are several model transformation languages that exist. Examples of these transformation languages are Atlas Transformation Language (ATL) [18], Query View Transformation (QVT) [20] and Triple Graph Grammars (TGG) [21].

2.7 The Ericsson DSL

Currently, Ericsson uses an in-house developed DSL to create applications for its Baseband Switches. The DSL uses graphical notations and is known as Hive to Ericsson. This DSL is built using UML and UML profiles. Their UML profile is known as Hive Profile and is divided into three major parts which serve three different purposes when it comes to creating applications(see figure 2.5). The first part is the behavior part, this models how the system being created should behave. The second part is the structural part which models how different parts of the system are structured and the relationships between different parts. The third part is the deployment part which models how the system should be deployed depending on the hardware being used. The Hive profilr in total is made up of 14 stereotypes from the Hive deployment profile, 17 stereotypes, 1 Enumeration and 1 class from Hive Behavior and 2 stereotypes from Hive Structure. The Hive DSL also uses some metaclasses from the UML metamodel as they are, .i.e. without extending them with any stereotypes.

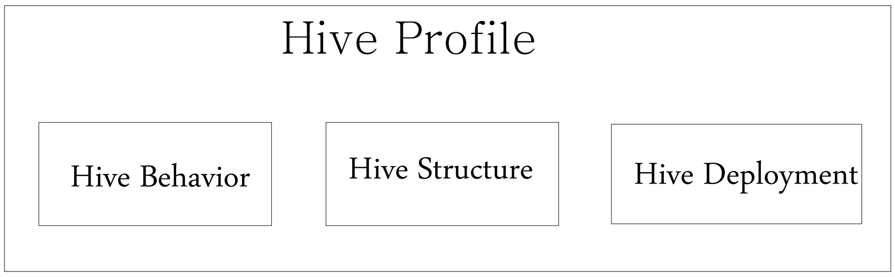


FIGURE 2.5: The Hive Profile composition.

At Ericsson, developers create models of the applications using this graphical DSL and later use model transformation tools to transform the models into C code (.c and .h files) which can be compiled into working applications. The transformations are done in three steps(see figure 2.6). The first step is a model to model transformation that transforms an instance model created using the Hive DSL to Dive which is another Ericsson in-house metamodel. From the Dive, another model to model transformation is done that transforms the Dive model into a C instance model that conforms to the C abstract syntax tree. From the C abstract syntax tree a model to text transformation is performed to get .c and .h text files.

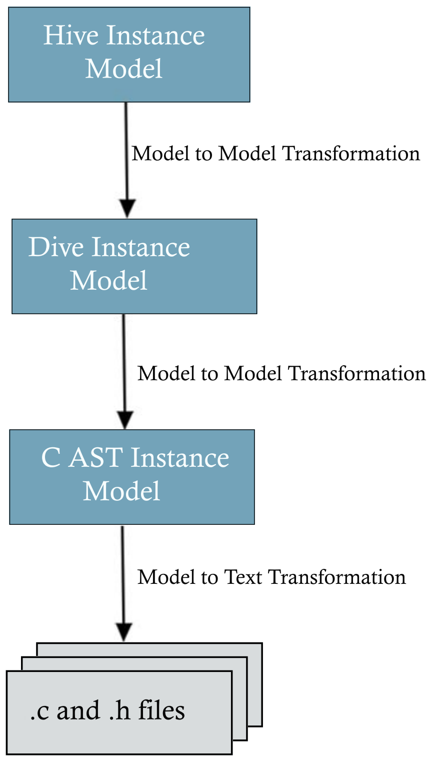

Hive Instance Model
Model to Model Transformation
Dive Instance Model
Model to Model Transformation
C AST Instance Model
Model to Text Transformation
.c and .h files

FIGURE 2.6: Code generation steps (From a Hive model to .c and .h files).

Chapter 3

Method

This thesis has been conducted using action research as its main research method. Action research is the type of research method where the researcher places him/herself in the organization where the problem is, studies the problem, implements a solution (by performing some actions) and analyze the impact of that solution in solving the problem [22]. From the results obtained the researcher can repeat the whole process again until the problem is fully solved or the researcher can propose a way forward to solving the problem.

The action research circle has five main stages. The first stage is diagnosing where the researcher and the practitioner come together and try to identify what the problem is. The second stage is action planning where the actions that need to be taken to solve the problem are planned. The actions are planned based on the hypotheses and research questions defined in the diagnosis stage. The third stage is action taking [2] or sometimes called intervention [22] where the actions planned are actually implemented. The fourth step is evaluation where the impacts of the implemented actions are analyzed. The fifth and final stage is reflection [22] also called specifying learning [2] where the researcher disseminates the knowledge to the organization and they together reflect on the impact and way forward. The figure below shows the action research cycle.

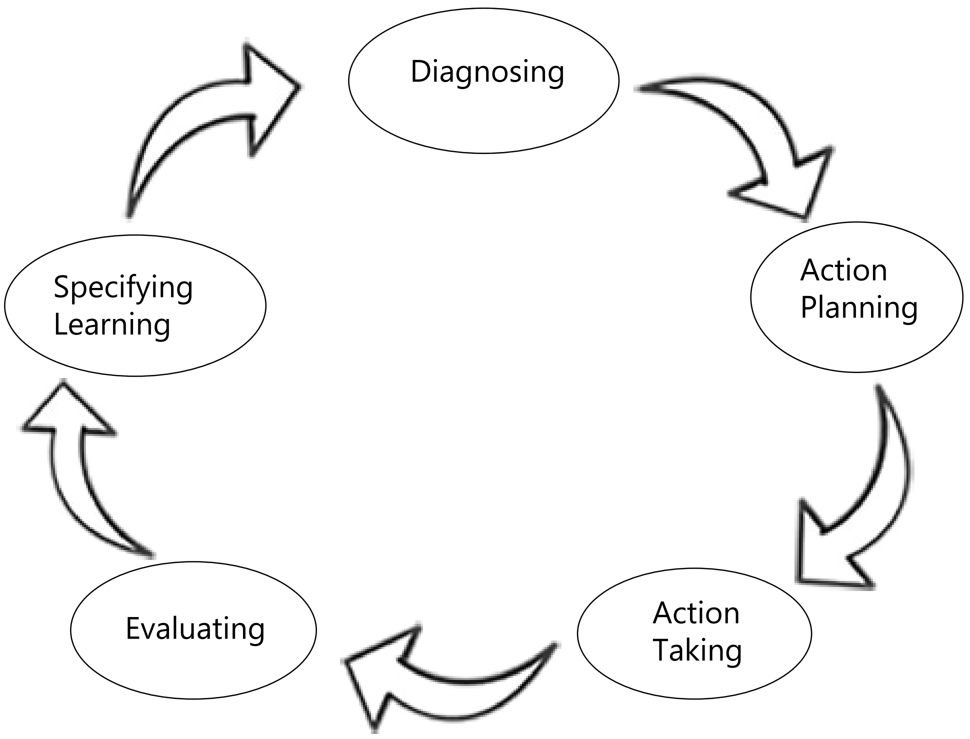


FIGURE 3.1: Action research cycle [2].

As mentioned in the introduction, Ericsson is the organization with the problem and as a researcher I was placed in the organization to study the problem, implement a solution and analyze the solution to see if it solves the problem. The thesis was conducted in four cycles of action research. The first stage of the cycles (diagnosis), started with a literature review on the problem. The papers selected for the literature review were selected by convenience sampling based on their relation to the subject and availability. The summary of the related work discovered during literature review is provided in chapter 4.

Then next was to understand what the actual problem was and to come up with research questions as well as hypotheses. This was done first by communicating with the concerned parties at Ericsson where they explained why having only a graphical version of a DSL was a problem and what the concerns were, when it comes to maintenance of two notations for the same DSL. Further analysis was conducted by getting the actual DSL used by the company(Hive), using it and knowing how it works. It is from these two initial steps where the research questions and Hypotheses were created. These research questions and hypotheses are listed below:

3.1 Research Questions

RQ1: What is the best way to have a DSL with both textual and graphical views without doubling the maintenance effort?

As it has been mentioned before, the company needs to add a textual version of the DSL but the addition of this DSL should not double the effort when it comes to updating the language and the transformation tool chain. This question aims to find out the best way (with lowest effort) to have textual and graphical representation of a DSL existing in a company.

RQ2: How can we switch between Textual and Graphical DSL without any loss of information?

Since there will be both graphical and textual versions of the DSL, there needs to be a good and easy way for programmers to switch between them. The aim here is to make the switching between the DSLs as automatic as possible.

Q3: How can syntax errors be handled when switching between them?

Models created by both graphical and Textual DSLs can be inconsistent. Inconsistent models are models that do not comply with the metamodel constraints. The aim of this question is to investigate whether syntax errors have any effect when one is switching between the two views.

3.2 Hypotheses

H1: Having a DSL with two notations leads to doubling of the effort required to maintain the DSL.

H2: Switching from one view of the DSL to another does not cause any loss of infomation from the model.

H3: Syntax errors cause problems when switching between graphical and textual views.

From the research questions above, three alternatives were discovered that would enable having a DSL supporting both textual and graphical views. It was planned then to do the implementations of these alternatives in cycles and analyze the impact of each. These alternatives and their implementation are discussed in detail in chapter 5.

The three alternatives were investigated in four cycles of action research. The first cycle was to implement alternative one for the behavior part of the Hive DSL, the second cycle was to implement alternative one again but for the structural part of the DSL. The reason for implementing the first alternative in two cycles is to be able to see if the solution will work for both cases. So the second cycle acted like a test to see if the solution can hold for other profiles as well. The deployment part was left out due to time constraints of the thesis. The third cycle was to implement a small prototype for alternative two of the solution and the fourth cycle was for alternative three of the solution. For each alternative an evaluation was done, findings were recorded and conclusions were drawn.

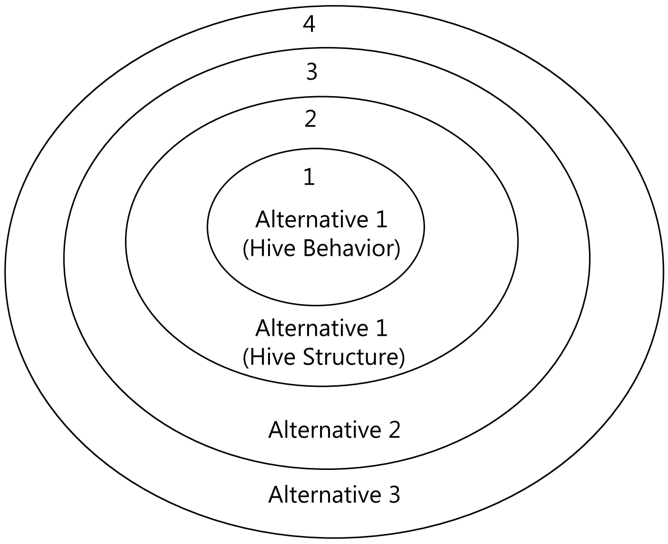


Figure 3.2: Cycles of Action research implemented.

Chapter 4

Related Work

From the literature review conducted, several approaches to provide a solution for a DSL with both graphical and textual views have been discovered. In [23], Colin Atkinson and Ralph Gerbig propose a technique that separates the actual notation of the DSL from its abstract syntax. They suggest the use of multi-level modeling and use of visualizers for editing models. A visualizer is a tool that determines how the models will be viewed as text, graphics or in tabular notation. Their concept of graphical editing is based on a technology similar to GMF while their concept of textual editing is based on projectional editing. Projectional editing is a technology that does not rely on a parser [24]. With projectional editing, the user edits the Abstract Syntax Tree(AST) of the model directly. The editor only shows a projection of what is in the AST and as the user is editing through the editor he/she is actually making changes to the AST of the model directly. Projectional editing has its drawbacks, as the user cannot save any kind of layout format for the text or graphics, when the model is saved only the AST of the model is saved. Reopening the model gives a default layout. This is a disadvantage if custom formats of the text or diagrams are of importance to the users. Projectional editing also does not support adding of comments in certain parts of code [24]. similar technology (projectional editing) has been used to create an Integrated Development Environment (IDE) known as mbeddr [an Extensible MPS-based Programming Language and IDE for Embedded Systems]. This IDE also supports the editing of a model in graphical, tabular and textual views but does not work on eclipse yet.

Another closely related work is that discussed in [25] which shows how to create a multi-view DSL. The idea here is to have one metamodel for the whole DSL and split it into several sub-metamodels according to various perspectives. The sub-metamodels are called viewpoints which can then have graphical or textual concrete syntax. Models are created using these viewpoints (either with graphical or textual notation) and later integrated in one repository. The repository is aware of how these sub models should be integrated to conform to the metamodel of the whole DSL. Also in this approach the textual language for a viewpoint is not manually created but rather automatically generated using Triple Graph Grammar Rules.This approach is different from our solution as our solution proposes a way to have textual and graphical notation for the whole DSL and not just part of the DSL.

A different approach is proposed in [11], where textual editors are embedded in graphical editors. This way when modeling using graphics, designers have an option to bring up a text editor that they can use to edit a model element that they have created in graphics.

When it comes to switching between graphical and textual views, [26] proposes two approaches to facilitate the transformation of models that are written in both graphical and textual notation (i.e. one model containing parts written in UML and parts in text). The first approach is called Grammarware and this simply refers to a text to text transformation of the models. With this approach the models are exported as text and transformation is done from text conforming to one metamodel to text conforming to another metamodel. The second approach is called Modelware and this refers to a model to model transformation. In this approach a model containing the graphical and textual content is transformed to a fully graphical model. This is done by converting the text part to its corresponding model element in the graphical metamodel.

This thesis however proposes a way to have a DSL supporting both graphical and textual views and the possibility to switch between them but not combining text and graphics in the same file. The thesis also investigates approaches of having both textual and graphical views which do not double tool maintenance efforts when it comes to updating the DSL. Moreover the thesis is conducted in a company where the existing DSL uses UML and UML profiles, so it also proposes a solution for UML based DSLs.

Chapter 5

Solution

From the related theory and other information obtained during the situation assessment phase, we came up with three alternatives that could be applied to get a DSL that supports both graphical and textual views. As mentioned in previous chapters, every DSL needs to have a metamodel. A metamodel defines the modeling concepts and determines which models are valid for a particular DSL [9]. The three alternatives are discussed below.

5.1 Alternative 1

The first alternative was derived from the current situation of the company. Ericsson currently uses a graphical DSL that is based on UML and UML profiles. Xtext on the other hand uses metamodels which are in Ecore format. So in order to obtain the textual version of the DSL a metamodel in Ecore has to be obtained first. It has been mentioned before, the company needs to add a textual version of the DSL but the addition of this DSL should not double the maintenance effort when it comes to updating the language. To avoid this, the Ecore metamodel is therefore derived from the existing UML Profile using model to model transformation. The idea being that once the language evolves, the changes will be applied to the UML Profile and the textual metamodel will be derived. The figure below illustrates this alternative.

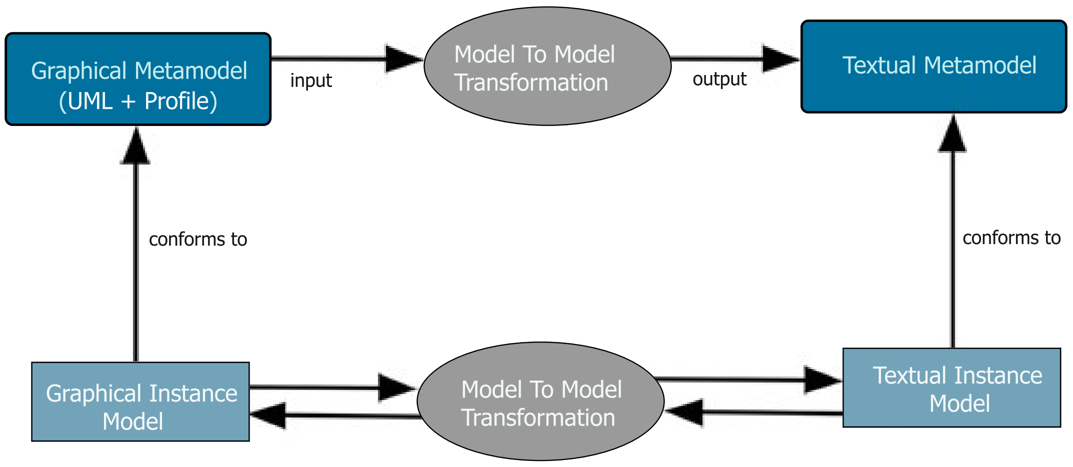


FIGURE 5.1: Alternative One.

The implementation of this alternative is discussed as a general solution. This means that this solution is not only valid for the Ericsson DSL but also valid for any other DSL that uses UML and UML profiles. The specific case for Ericsson's DSL is discussed in Case 1 and Case 2 chapters. The first step towards this alternative is to obtain a metamodel in ecore that can be used to generate our textual editor. How this metamodel is obtained is explained below:

5.1.1 UML Profile to a Metamodel in Ecore

The Eclipse Modeling Framework comes with a functionality that can export UML models into Ecore models. Using this functionality was the first approach to obtain a metamodel in Ecore from the UML Profile. This functionality worked well but had a huge drawback since it also exports the whole UML metamodel to the exported Ecore model. When generating the grammar from this Ecore, every element from the Ecore plus the UML metamodel is generated in the grammar. This makes the grammar very huge with a lot of unused elements. This increases the amount of effort required to maintain a huge textual language while in many cases people only use a small part of the UML metamodel.

To be able to solve this problem, a subset of classes from UML that are actually needed for the DSL needs to be obtained.

Ways to obtain the UML Subset:
To be able to obtain the UML Subset, one needs to know exactly which classes are used. These classes include those extended by the stereotypes in the profile and also those that are used without any stereotypes. For some DSLs this set of classes is known and for

some DSLs it may not be so obvious which classes are used. This is especially when users use part of UML that is not extended by any stereotype in the profile. In such cases this list of classes needed can be obtained by running a transformation that takes an instance model of the DSL and returns a collection all UML metaclasses used on that instance model. This will give a correct list of classes needed if the instance model covers 100% of the DSL. In case no such instance models exist, one can identify the needed UML metaclasses manually and create a list of these classes either as an Ecore or as another UML profile that will only be used to identify these metaclasses. Once these classes have been identified a transformation can be written that copies only these classes from UML to create a subset UML metamodel. This UML subset can also be created manually as an Ecore model that contains all the classes of the subset and their attributes. However if the DSL changes frequently then this subset can be hard to maintain.

With the UML subset, another transformation needs to be created which takes the UML Profile and UML subset as input and produces a model in Ecore that can be used in Xtext to generate grammar. This transformation can be written using any transformation language. The mapping used to convert UML to Ecore follow the ones used in the UML to Ecore eclipse plugin [27] and also according to the relationship between UML and Ecore as described in [15]. These mappings are described below:

UML Profile to Ecore Package: Every profile is transformed to a corresponding Ecore Package. Since a profile contains Stereotypes as its sub packages, when these stereotypes are transformed they will be nested elements to the corresponding package.

UML Stereotypes to Ecore classes: UML Profiles contain Stereotypes and tagged values. Stereotypes are the additional UML classes while tagged values are the additional attributes and references. A stereotype extends a metaclass, which is a class that already exists in UML. To transform a stereotype to Ecore first transform the extended UML metaclass to an EClass in Ecore, with all the properties of the class as Attributes in Ecore and all references as Ereferences in Ecore. Then transform the stereotype to an EClass in Ecore and all its tagged values as EAttributes in Ecore. To maintain the relationship between the metaclass and stereotype in the Ecore model as well add a reference in the EClass corresponding to the metaclass to the EClass corresponding to the stereotype. This reference represents the extension relationship in UML. The figure below illustrates this.

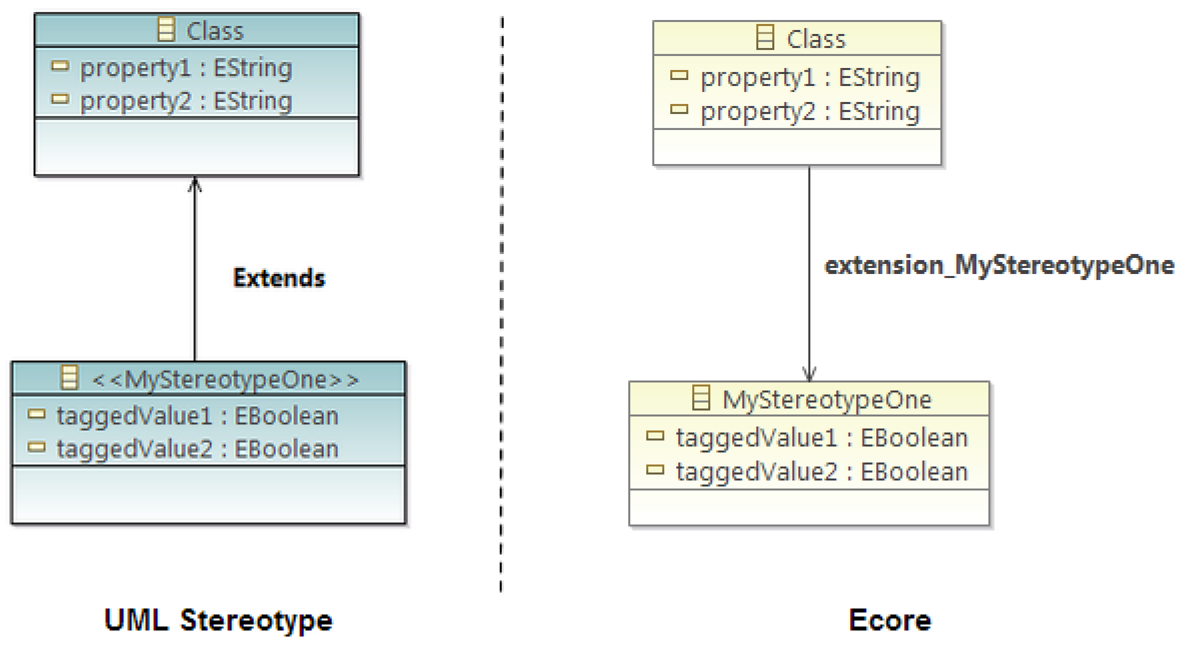


FIGURE 5.2: UML Stereotypes to Ecore classes.

UML Metaclasses to Ecore Classes: UML metaclasses are those classes that are present in the UML metamodel. It has already been established that it is better to use a subset of UML rather than the whole UML metamodel unless one is using all the classes from the metamodel in their DSL. So in this case the UML metaclasses will be from the UML subset and not the complete UML models. These classes are transformed to EClasses in Ecore. If this subset of UML was supplied in form of Ecore already then these Eclasses are just copied to EClasses in the output model as well. And in this case the transformation will take in the UML profile conforming to the UML metamodel and a UML subset conforming to the Ecore metamodel as input and produce an Ecore model conforming to the Ecore metamodel as output.

UML Property to EAttributes: Each UML property whose type is a primitive data type (String, Integer, Boolean or Real) is transformed to an EAttribute in Ecore and the corresponding data type (EString, EInt, EBoolean or EDouble). This is done for all the properties from the Stereotypes.

UML Property to EReference: Each UML property, whose type is another class, is transformed to an EReference in Ecore. And the type of this EReference is set to the corresponding class in Ecore. For instance if the property in UML is called Base and its type is Activity, then the property will be transformed to a reference called Base and the type will be a class named Activity in Ecore. This is done for all the properties from the Stereotypes.

UML Data Type to Ecore Data type: In UML a Data Type is a class that is used to define certain kinds of data values. An example of a UML Data Type is Date. In the transformation each UML Data Type in the profile is transformed to an EDataType in Ecore. This applies to all Data Types except Enumeration Data Type.

Enumerations to EEnum: All enumerations in UML Profile are transformed to EEnum in Ecore. The corresponding Enumeration literals are transformed to EEnum literals in Ecore. It is important to note that the transformed EEnum should also be in the same package as all the transformed Stereotypes.

The table below gives a summary of the mappings from UML Profile to an Ecore model:

UML	**Ecore**
Profile	EPackage
Stereotype	EClass
Metaclass	EClass
Property (with Primitive type)	EAttribute
Property (with other classes as type)	EReference
Data Type	EDataType
Enumeration	EEnum

Table 5.1: UML to Ecore Mapping

The listing in the table above does not cover all the UML model elements. It only contains the necessary model elements which are needed when transforming a UML profile to Ecore. Model elements which are not listed here are UML Operations, Map References and UML Documentation [15]. The code for the transformation of a UML Profile into an Ecore model is included in Appendix B of this report.

5.1.2 Generating the Textual Editor using Xtext

From chaper 5.1.1 above, once this Ecore model has been obtained it can be used in Xtext to generate the Textual concrete syntax of the DSL. This is done by creating an Xtext project from an import of the Xtext Ecore metamodel. This automatically generates a grammar for us. In most cases this grammar is not satisfactory for users as it follows one template for all model elements. So the user has to do some manual editing to get the grammar to look like what is desired. It is also important to note that the auto generated grammar generates blank rules (only titles for the rules) for EEnums so contents of these grammar rules have to be manually added by the user. Once the grammar is complete the textual editor can be auto generated with Xtext. Having a textual editor from Xtext means that now it is possible to create models using text. The next step is to enable the switch between textual and graphical views. This is elaborated below:

5.1.3 Switching between Graphical and Textual Views

5.1.3.1 Xtext Instance Model to UML Instance Model

Once the XText editor has been created it can be used to create textual instance models. These instance models can then be transformed to UML instance models. Note that the Xtext instance model can be serialized to an XMI file so that it can be used in transformations. The transformation from Xtext instance model to UML instance model can be written in any model to model transformation language, but the mappings should be as described below. To facilitate better understanding examples of code for transformation rules written in ATL is given.

a. Classes Matching UML Metaclasses with no Associated Stereotypes
Instances of classes that match the UML metaclasses and have no associated stereotype in the profile used are transformed to their corresponding instances of UML Metaclasses. Associated stereotype in this case means that the profile used contains one or more Stereotypes extending the UML metaclass. For instance if there is an instance of a class called Activity in Xtext instance model, and the profile used does not contain any stereotype extending the Activity UML metaclass, this instance is transformed to an instance of the UML metaclass called Activity. The attributes values that are present in the Xtext instance model are also copied to the corresponding attributes in UML.

```
 1 rule ActivitytoActivity {
 2
 3   from s:Xtext!Activity
 4
 5   to t:UML!Activity (
 6
 7   name <- s.name,
 8   node <- s.node
 9
10
11   )
12
13 }
```

Listing 5.1: An ATL matched rule showing a transformation from Xtext Activity to UML Activity.

The listing above shows an ATL rule illustrating this kind of transformation mapping. In the figure an instance of an EClass of type Activity from Xtext is transformed to a UML instance class of type Activity (see line 3 and 5). The attributes and references of this activity instance from Xtext are also copied to the target UML activity instance. In this case the instance of type Activity from Xtext has one attribute called name (see line 7) and a reference called node (see line 8).

b. Classes Matching UML Metaclasses and have Associated Stereotypes
Those classes that match the UML Metaclasses and have associated stereotypes in the UML profile used are transformed to UML Metaclasses that they conform to. For instance if a class called Operation has one or more stereotypes in the profile extending it, then this class will be transformed to UML metaclass called Operation and all its attributes and references copied to it. The second step is be able to apply stereotypes to the newly created UML Class. In order to know which stereotype to apply, first one has to check which extension reference is associated with this class. If the class has more than one extension reference it means that there are several stereotypes that extends this metaclass and a proper check is needed for which stereotype needs to be applied. For example if this particular instance of operation has an extension reference of type HiveMapToActivity, then the stereotype called HiveMapToActivity will be applied and the corresponding tagged values copied. It is important to also note that there are cases where more than one stereotype needs to be applied to a class.

```
rule OperationToOperation {
  from
    s:Xtext!Operation
  to
    t:UML!Operation (

      name <- s.name,
      visibility <- s.visibility,
      type <- thisModule.getDataType(s.type.toString())

    )

    do {

      if (s.extension_HiveBaseMapToBehavior.oclIsTypeOf(MM!HiveMapToActivity)) {

        t.applyStereotype(thisModule.HiveMapToActivityStereotype);


        if (thisModule.hasValue(s.extension_HiveBaseMapToBehavior.activity))
        {
          t.setValue(thisModule.HiveMapToActivityStereotype, 'activity', thisModule.getActivity(s.extension_HiveBaseMapToBehavior.activity));
        }

25          if (thisModule.hasValue(s.extension_HiveBaseMapToBehavior.event))
26          {
27            t.setValue(thisModule.HiveMapToActivityStereotype, 'event',
     thisModule.getEvent(s.extension_HiveBaseMapToBehavior.event));
28          }
29        }
```

Listing 5.2: An ATL matched rule to transform an Xtext Operation to UML Operation.

The code in the listing above illustrates an example where an instance of type operation from Xtext is transformed to a UML instance of type operation (see line 3 to 5). The code also checks to find out which stereotype needs to be applied to the UML Operation. In this case we check if this instance of Operation from Xtext contains an extension known as HiveMapToActivity. If yes then the HiveMapToActivity stereotype is applied to the resulting instance of UML Operation. When the stereotype has been applied the tagged values need to be set as well. This is done from line 20 to line 27 where we first check if the attributes are not null and later copy their values using the setValue function.

5.1.3.2 UML Instance Model to XText Instance Model

Instance models in graphical format created using UML can also be transformed to models conforming to the XText metamodel. The mapping of this transformation is as follows:

a. Classes with No Stereotypes Applied

Instances of classes that do not have any stereotype applied are transformed to instances of matching classes in the Xtext Ecore model. For instance if in UML there is an instance of a class called Transition, then this will also be transformed to correspond to an instance of the Transition class in the XText instance model. Knowing that the XText instance model was created by copying the classes from UML, then this transformation is also just a copy from a UML class to a corresponding class in XText. The properties of the classes are also copied to the corresponding attributes in Ecore. In case of attributes that are compulsory in XText but not supplied in the UML instance model, then default values can be set using the transformation.

The listing below shows an example of transformation from UML transition that has no stereotype applied. In this case the transition has a total of 7 attributes and references to be copied which are name, container, kind, redefinedTransition, source, target and trigger.

```
rule TransitionToTransition {

  from s:UML!Transition ( s.getAppliedStereotypes() -> isEmpty())

  to t:Xtext!Transition (

    name <- thisModule.getTransitionName(s),
    container <- s.container,
    kind <- s.kind,
    redefinedTransition <- s.redefinedTransition,
    source <- s.source,
    target <- s.target,
    trigger<- s.trigger

  )
}
```

LISTING 5.3: An ATL matched rule to transform a UML Transition to Xtext Transition.

b. Classes with Stereotypes Applied

Instances of classes that have stereotypes applied are transformed to two instances in Xtext. First to an instance of a class conforming to the UML metaclass that the stereotype extends and then to another instance that conforms to the applied Stereotype classes that are in the Xtext Ecore. A reference is also created from the metaclass EClass to the Stereotype EClass. For instance if a class has a stereotype called HiveMapToActivity applied and this stereotype extends a metaclass called Operation then it will be transformed to an EClass called Operation in the Xtext instance model and an EClass called HiveMapToActivity in the XText metamodel. A reference will be created from the instance of Operation to the stereotype HiveMapToActivity. The tagged values are then copied from the UML stereotyped class to the corresponding attributes in the XText instance of the class. The listing below illustrates these mappings with ATL code. Note that in this example extension_HiveBaseMapToBehavior is the reference from the class Operation to the class HiveMapToActivity. 'activity' and 'event' are tagged values of the HiveMapToActivity stereotype.

```
rule HiveMaptoActivityToOperation{

  from s:UML!Operation (s.isStereotypeApplied(thisModule.
    HiveMapToActivityStereotype) )

  to t:MM1!Operation (

    name <— s.name,
    type <— s.datatype,
    extension_HiveBaseMapToBehavior <— t2
  ),

  t2: Xtext!HiveMapToActivity (

    activity <— s.getValue(thisModule.HiveMapToActivityStereotype, '
    activity'),
    event <— s.getValue(thisModule.HiveMapToActivityStereotype, 'event')

  )
}
```

LISTING 5.4: An ATL matched rule showing a transformation from UML Operation to Xtext Operation.

5.1.3.3 Higher Order Transformations

Model transformations can either be manually written or automatically generated using other transformations. A transformation that produces other transformations as output is known as a Higher Order Transformation (HOT). Higher Order Transformations are possible since transformations are also models that conform to a certain transformation metamodel [4]. For instance an ATL Transformation is a model that conforms to the ATL metamodel. A HOT either takes any model as input or another transformation as input to produce other transformation(s) as output.

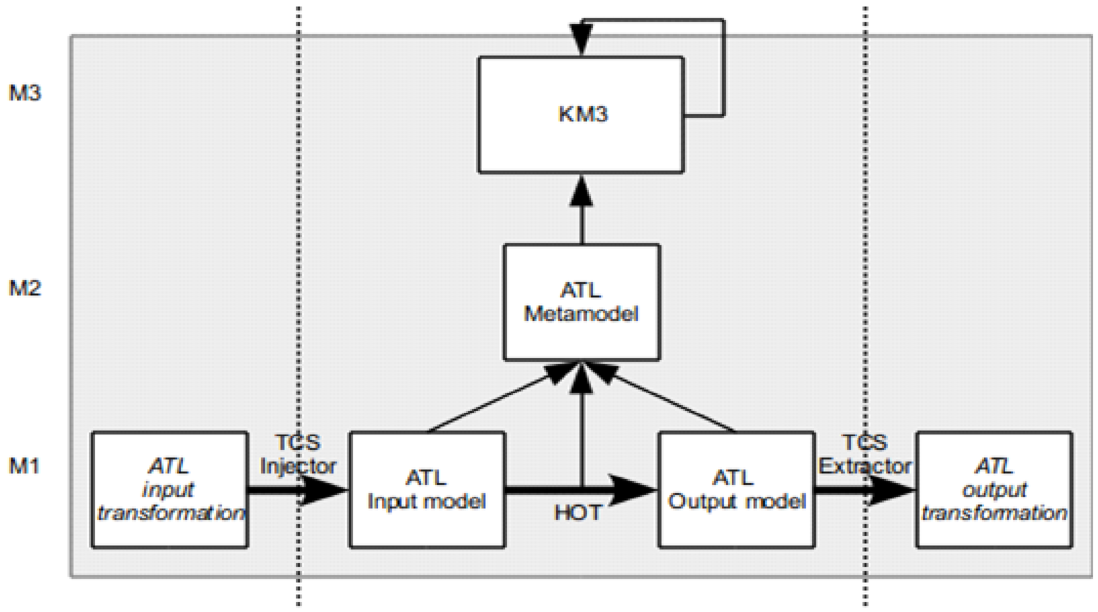


FIGURE 5.3: ATL Higher Order Transformation Chain[3].

The figure above illustrates the concept of HOT. From the left hand side a transformation (ATL input transformation) is first serialized as a model conforming to the ATL metamodel using the Textual Concrete Syntax (TCS) Injector [28] and the model is called ATL input model. This model is given as input to a HOT which produces the ATL Output model conforming to the ATL metamodel as its output. The TCS Extractor is used to serialize the ATL Output model to the ATL textual syntax and the final output which is a transformation in ATL textual syntax is called ATL output transformation.

With this concept, the transformations from UML instance model to Xtext instance model and from Xtext instance model to UML instance model do not have to be written manually. Instead HOTs can be used. Since we know that the metamodel for the Xtext language is generated from the UML profile and UML subset, then all information needed for our instance model transformations can be obtained from these two models (UML Profile and UML subset). We therefore need to write a HOT that takes the UML Profile and the UML subset as input and produce an instance model transformation as output.

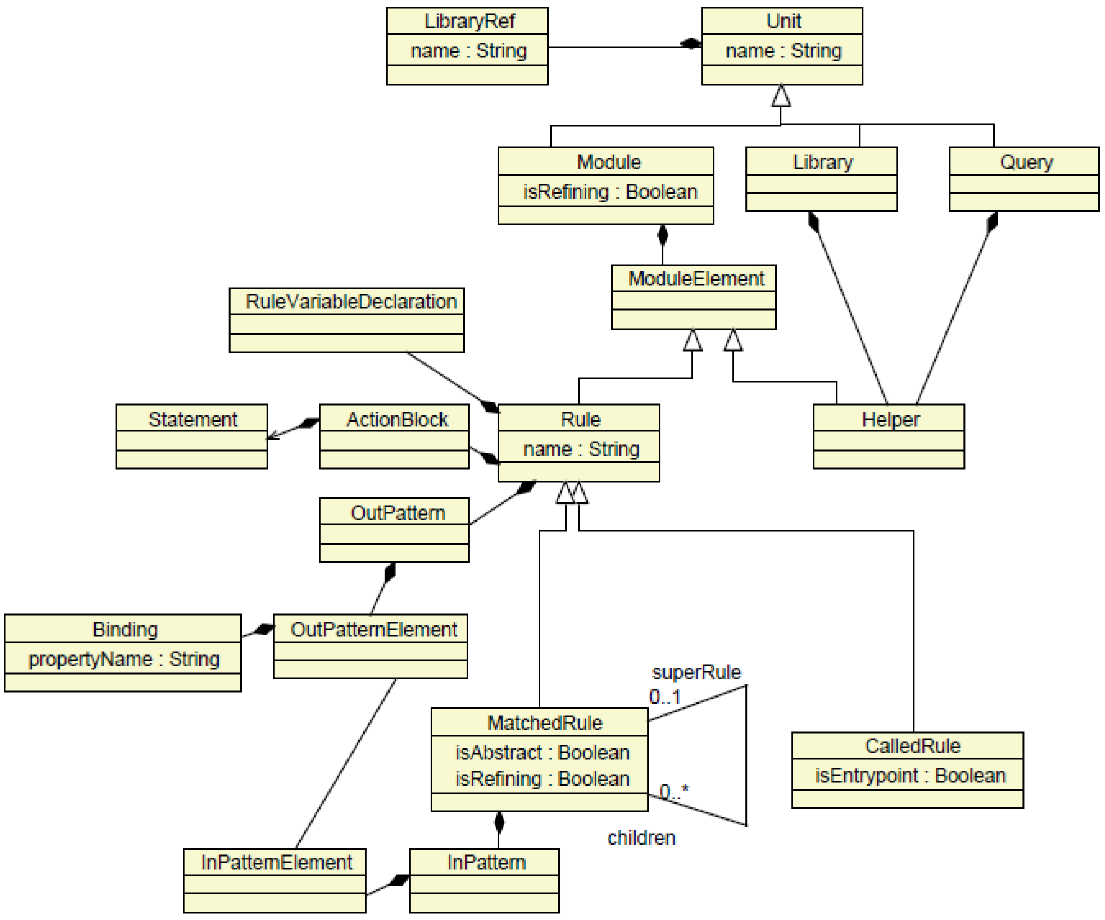


FIGURE 5.4: Simplified Version of the ATL metamodel[4].

To generate bindings of the transformations attributes of the UML profile and UML subset can be used. This works if the bindings have a one to one relationship from UML to Xtext and vice versa. If the bindings from UML to Xtext are not one to one mappings, the HOT transformation needs more information in order to create these bindings. A trace model can be used as another input to the HOT in order to facilitate this. A trace model is a model that defines the relationship between the source model and the target model. It describes how each model element in the source model should be realized as a target element in the target model. The figure below shows a trace metamodel. From the metamodels, one starts to define a trace model by creating a Link set. The link set can contain several trace rules in it. A traced rule contains links, these links contain source elements and target elements. The source element will be mapped to the target element in the generated transformation.

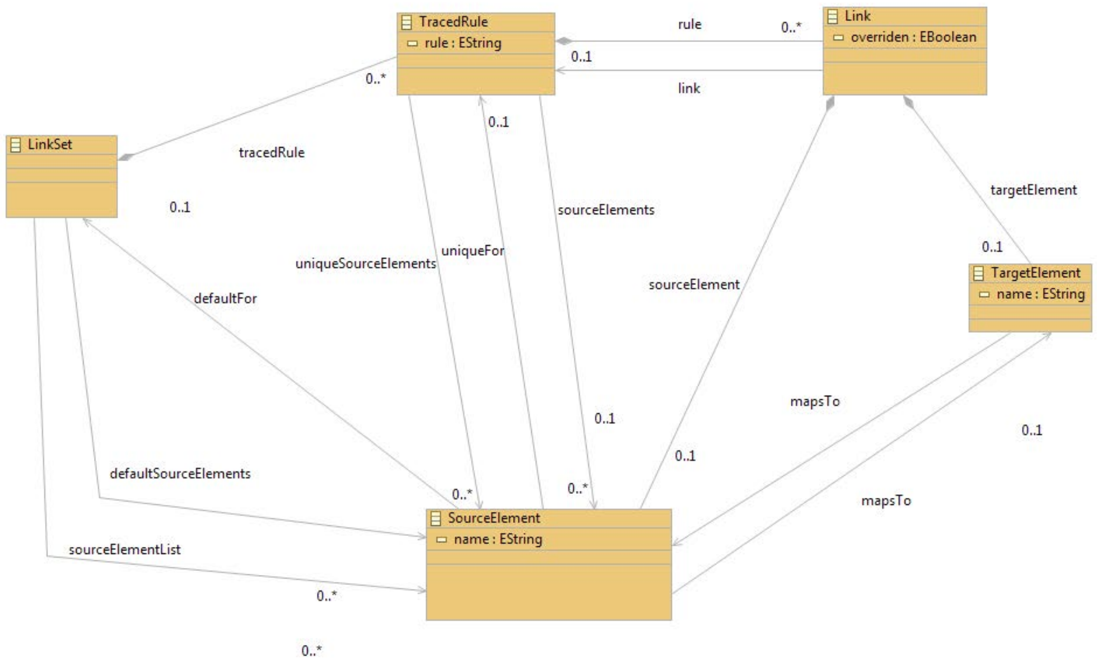


FIGURE 5.5: Simplified Version of the ATL trace model [4].

Assume you have two metamodels each with one class in it. Metamodel A with a class called ElementA with two attributes, name and type. Metamodel B with class called ElementB with two attributes uniqueId and kind. If you want to generate a rule that transforms Element A to Element B you need to specify the relationship between these two metamodels in a trace model. For this case your trace model will contain a link set with one traced rule (see figure 5.6). The traced rule will contain two links, one for each of the attributes. In the first link we define what is the source attribute (in our case name) and what is the target attribute (in our case uniqueId). The same is done for the second link in which the source attribute is type and the target attribute is kind.

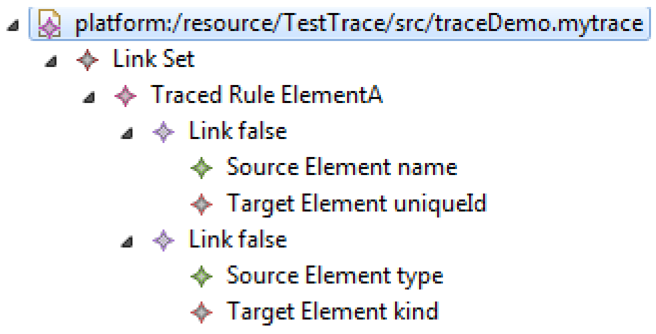

platform:/resource/TestTrace/src/traceDemo.mytrace
Link Set
Traced Rule ElementA
Link false
Source Element name
Target Element uniqueId
Link false
Source Element type
Target Element kind

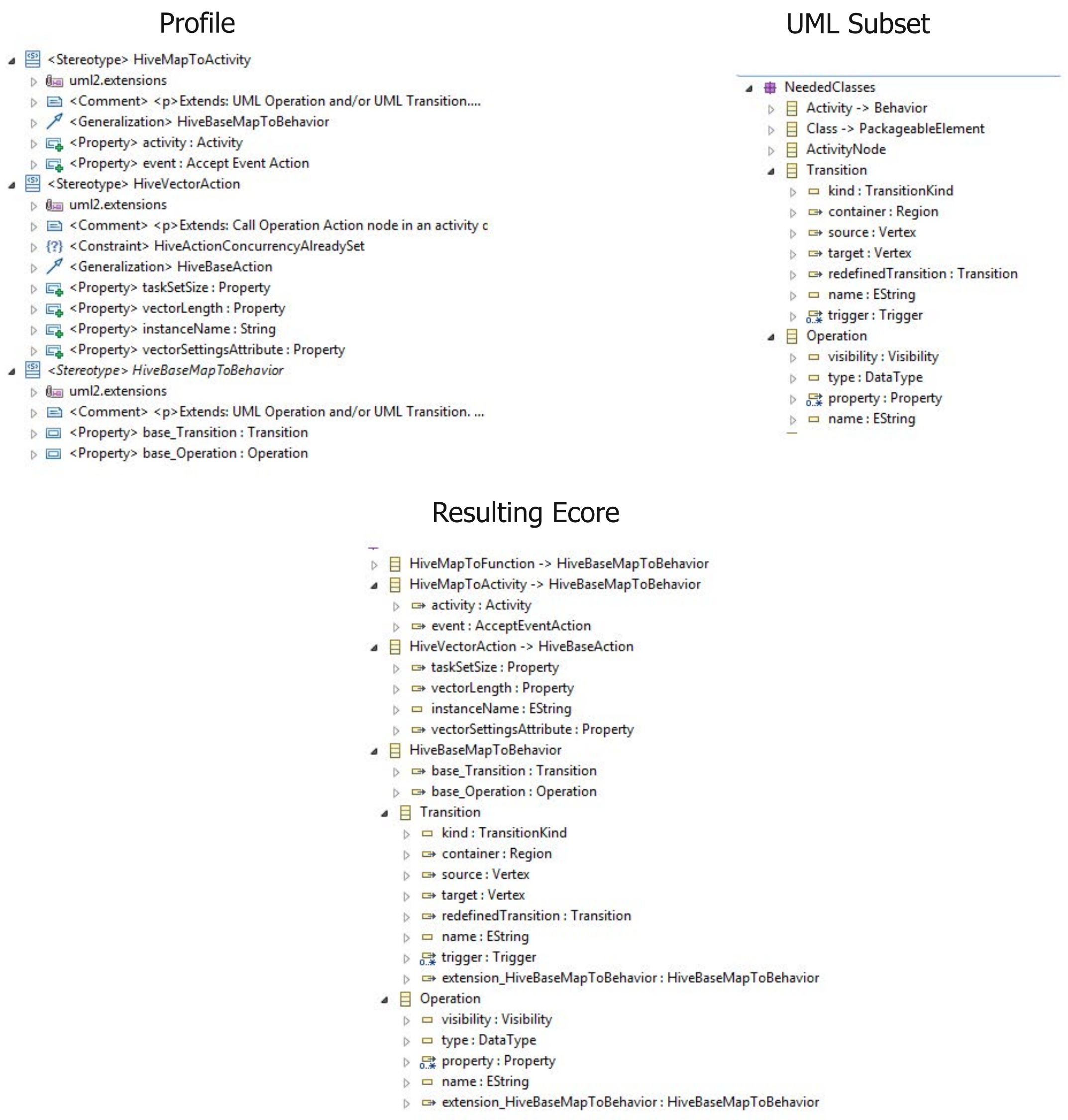
FIGURE 5.6: Sample trace model.

5.1.4 Case 1

The solution alternatives described were applied to the Ericsson case. As it has been mentioned the DSL at Ericsson is a graphical DSL that uses UML and UML profiles. The DSL can also be divided into three parts namely Behavior part, Structural part and Deployment part. The first case was to apply the described alternatives for the behavior part of the DSL and create Hive Behavior DSL supporting both graphical and textual views. The hive behavior profile is made up of 17 stereotypes, 1 Enumeration and 1 class.

5.1.4.1 Metamodel Transformation

The first step in applying the solution was to obtain a metamodel in Ecore so that we can use the metamodel to generate Xtext grammar. For the generated grammar to be small and concise a subset of the UML metamodel is needed as well (as discussed in chapter 5.1.1). For the case of Ericsson there is no formal definition of which classes are actually used from the UML metamodel so this information had to be obtained manually. It was obtained by looking through the instance models that already existed and identifying the used classes and attributes. Then an Ecore containing these used classes and their attributes was created. Those classes from UML that were used with the Hive profile but did not make sense in the textual language were omitted. An example in this case is the activity edge class, when defining activities using diagrams drawing edges like control flows is a good thing but when it comes to specifying activities using text, the edges add complexity to the language. So the activity edge class was not included in the UML subset and instead a "depends on" reference was introduced to each activity node except the initial node. The resulting UML subset contained 25 model elements from the UML model which is just 9.4% of the UML metamodel. The UML metamodel contains 265 model elements.Model elements refers to the metaclasses and Data Types contained in the UML metamodel.



Profile

```
<Stereotype> HiveMapToActivity
    uml2.extensions
    <Comment> <p>Extends: UML Operation and/or UML Transition....
    <Generalization> HiveBaseMapToBehavior
    <Property> activity : Activity
    <Property> event : Accept Event Action
<Stereotype> HiveVectorAction
    uml2.extensions
    <Comment> <p>Extends: Call Operation Action node in an activity c
    <Constraint> HiveActionConcurrencyAlreadySet
    <Generalization> HiveBaseAction
    <Property> taskSetSize : Property
    <Property> vectorLength : Property
    <Property> instanceName : String
    <Property> vectorSettingsAttribute : Property
<Stereotype> HiveBaseMapToBehavior
    uml2.extensions
    <Comment> <p>Extends: UML Operation and/or UML Transition. ...
    <Property> base_Transition : Transition
    <Property> base_Operation : Operation
```

UML Subset

```
NeededClasses
    Activity -> Behavior
    Class -> PackageableElement
    ActivityNode
    Transition
        kind : TransitionKind
        container : Region
        source : Vertex
        target : Vertex
        redefinedTransition : Transition
        name : EString
        trigger : Trigger
    Operation
        visibility : Visibility
        type : DataType
        property : Property
        name : EString
```

Resulting Ecore

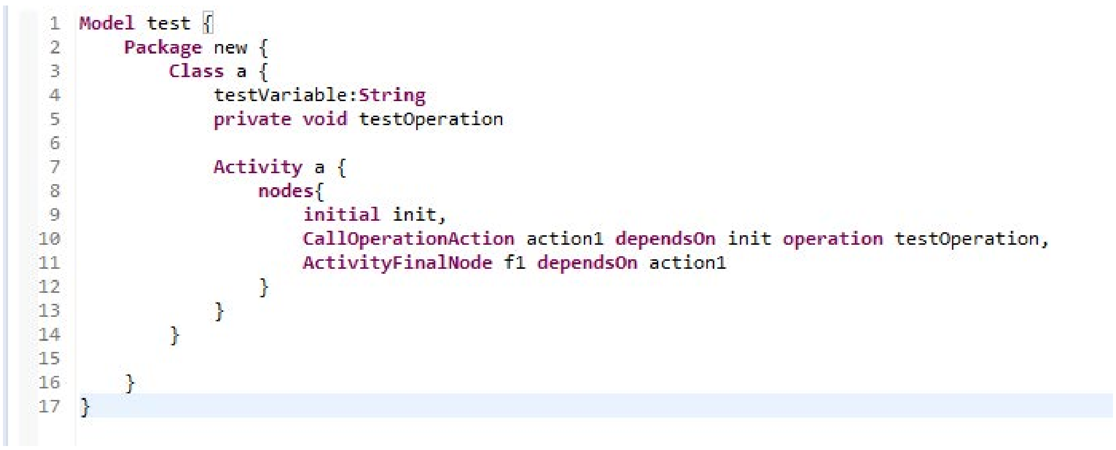
```
HiveMapToFunction -> HiveBaseMapToBehavior
HiveMapToActivity -> HiveBaseMapToBehavior
    activity : Activity
    event : AcceptEventAction
HiveVectorAction -> HiveBaseAction
    taskSetSize : Property
    vectorLength : Property
    instanceName : EString
    vectorSettingsAttribute : Property
HiveBaseMapToBehavior
    base_Transition : Transition
    base_Operation : Operation
 Transition
    kind : TransitionKind
    container : Region
    source : Vertex
    target : Vertex
    redefinedTransition : Transition
    name : EString
    trigger : Trigger
    extension_HiveBaseMapToBehavior : HiveBaseMapToBehavior
 Operation
    visibility : Visibility
    type : DataType
    property : Property
    name : EString
    extension_HiveBaseMapToBehavior : HiveBaseMapToBehavior
```

FIGURE 5.7: Part of the Hive Behavior Profile, its UML subset and the resulting Ecore after transformation.

A model to model transformation was written that takes the Hive Profile and the Ecore of UML subset as input and produces an Ecore model to be used in Xtext. The mappings from the Profile and UML subset are as mentioned in chapter 5.1.1. The figure above shows part of the Hive Behavior Profile, its UML subset and the resulting Ecore after transformation. In the figure one can see that the stereotype HiveMapToActivity from the profile has been transformed to a class called HiveMapToActivity in the resulting Ecore. The tagged values (Properties) from the HiveMapToActivity stereotype which are activity and event have been transformed to references in the HiveMapToActivity class in the resulting Ecore. The HiveBaseMapToBehavior stereotype has been transformed to a class called HiveBaseMapToBehavior in the resulting Ecore. Since this stereotype extends the Transition and Operation UML metaclasses references to this class are added in the Operation and Transition classes from the UML subset. These references are 'extension_HiveBaseMapToBehavior' in both the Transition and Operation classes. Opposite references for these are added in the HiveBaseMapToBehavior class. These references are 'base_Transition' and 'base_Operation'as seen in the figure.

After the metamodel was obtained, it was supplied to Xtext and Xtext generated the concrete syntax for the DSL. The generated concrete syntax needed some manual editing to make the syntax better. The editing done was mainly to remove unwanted keywords, curly braces and commas. Also editing was made to add contents for Enumerations since Xtext does not generate this. After editing the grammar, an Xtext editor and parser were generated. Using this editor it was now possible to create models using text.

A sample of a model created using the editor is provided in the figure below.



```
Model test {
    Package new {
        Class a {
            testVariable:String
            private void testOperation

            Activity a {
                nodes{
                    initial init,
                    CallOperationAction action1 dependsOn init operation testOperation,
                    ActivityFinalNode f1 dependsOn action1
                }
            }
        }

    }
}
```

Figure 5.8: Model created with Xtext editor.

5.1.4.2 Instance Model Transformations

Having the XText editor in place, one can now create instance models using text. The second problem that arises is how one can be able to switch between a model created in text to a model in graphical format (UML in this case). To facilitate this, first the instance model created using XText needs to be serialized as an XMI model. This enables the model to be used as input to an ATL transformation as the plain text format is not recognized with ATL. An ATL transformation was also written that takes this Xtext instance model and the Hive profile as input and produces a UML model with the Hive Profile applied. To enable switching from a UML instance model to XText instance model, another ATL transformation was also written. This transformation takes a UML model created using the graphical DSL (Hive) and the Hive profile as input and produces a model conforming to the XText Ecore Metamodel. The resulting model is an XMI model conforming to the Xtext metamodel and it can later be serialized to the textual concrete syntax in XText. This serialization could also be automated so that the user gets the resulting file in Xtext concrete syntax directly.

Looking at the instance model transformations written, it was discovered that most of the mappings in the transformations were one to one mappings. This meant that a huge part of the transformations could be generated using HOT instead of being manually written. The implementation of the HOT needed to generate these transformations is discussed in the following subsection.

5.1.4.3 Generating Transformations using HOT

As mentioned previously, ATL offers a functionality known as Higher Order Transformation (HOT), with this, one can write a transformation that produces other transformations as output. This is an advantage as instead of writing transformations manually one can write a transformation that generates another transformation. Given that there is a need to have two transformations for the instance models, it would be of value if these transformations were generated instead of manually written. Trying this approach with the Ericsson DSL led to the following discoveries. Parts of the transformations that had one to one mappings between UML and Xtext could be easily generated. This is done by writing a transformation that takes the Hive Profile and UML subset as an input and generates an ATL transformation as output.

To generate a transformation from UML to Xtext the HOT written needs to produce the following output. First we need ATL matched rules that will transform instances of UML classes with no stereotypes applied to instances of classes conforming to the metamodel used in Xtext. Second we need ATL matched rules that will transform instances of classes that have stereotypes applied to instances of classes conforming to the metamodel used in Xtext. Third within the ATL matched rules we need to have bindings that determine how the attributes and references from the source classes are related to the attributes and references of target classes. The generated transformation also needs to have helpers that identify each stereotype by name from the Hive profile. The implementation of a HOT that generates the above is described below:

- Generating an ATL Matched rule to transform instances of classes with no stereotypes applied:
  A matched rule like the one shown in listing 5.5 below is what we aim to generate.

This is an example of a rule that will transform an instance of a UML Activity to an instance of Xtext activity.

```
1 rule Activity2Activity {
2  from
3   s : UML!Activity
4  to
5   t : XTEXT!Activity (
6    name <- s.name,
7    node <- s.node
8   )
9 }
```

Listing 5.5: An ATL matched rule generated from UML Subset

The generation of a rule like in listing 5.5 is done by taking every class from the UML subset used and from this UML subset generate one ATL matched rule. So if one of the classes is called Activity then a rule will be generated that will take an instance of an activity class in UML and generate an instance of an Activity class in Xtext(see listing 5.5).

The ATL HOT rule to generate this is given in listing 5.6 below. Line 3 of the rule means that we take the UML subset in our case called (UMLSub) and all classes from this subset. Then from each of these classes we generate one ATL matched rule (see line 5 of the HOT rule in listing 5.6). The generated rule will have a name which will be of the format 'n' 2 'n' where 'n' represents the class names (see line 6 of listing 5.6). An example of the results from this line is line 1 on listing 5.5 where the name of the rule is Activity2Activity. The generated rule will also have an inPattern and outPattern (see line 7 and 8 of listing 5.6).

The inPattern in our case will be of UML type represented by a variable named 's' (see line 11 to 29 of listing 5.6). The output of this is line 3 of listing 5.5. The outPattern of the generated rule will be of Xtext metamodel type represented by a variable named 't' (see line 31 to 37 of listing 5.6). The output of this is shown on line 5 of listing 5.5 where XTEXT!Activity represents an Activity type in Xtext. The output pattern also needs to have bindings that specify how the attributes and references of the source class are related to the attributes and references of the target class. These bindings are created in line 38 of listing 5.6. Since the bindings are one to one bindings we fetch all attributes and references from the source classes and create similar bindings for them in the target classes. An example of these bindings created can be seen on listing 5.5 line 6 to 7. In this case the Activity class had one attribute called 'name' and one reference called 'nodes', hence the two bindings.

```
rule classesFromUMLSubset {

  from s:UMLSub!EClass

  to t:ATL!MatchedRule (
    name <- s.name + '2' + s.name,
    inPattern <- inpat,
    outPattern <- outpat
  ),

  inpat: ATL!InPattern(
    elements <- elementIn

  ),

  elementIn :ATL!SimpleInPatternElement (
        varName <- 's',
        type <- t1
        ),

  t1 : ATL!OclModelElement(
      name <-  s.name,
      model <- m
      ),

 m : ATL!OclModel (
      name <- 'UML'

      ),

  outpat:ATL!OutPattern (
    elements <- elementOut
  ),

  elementOut:ATL!SimpleOutPatternElement(
    varName <- 't',
    type <-  t2,
    bindings <- UMLSub!EClass.allInstancesFrom('IN1')->select(e|e.name = s.
   name)->collect(e|e.eStructuralFeatures)
  ),

  t2 : ATL!OclModelElement(
      name <-  s.name,
      model <- m2
      ),
 m2 : ATL!OclModel (
      name <- 'Xtext'
      )
}
```

Listing 5.6: HOT rule t generate an ATL matched rule

- Generating an ATL Matched rule to transform instances of classes with stereotypes applied:

  The expected output rule from our Higher order transformation is a matched rule like the one shown in listing 5.7 below.

```
rule HiveMapToFunctionStereotypedClass {
 from
  s : UML!Transition
  (
   s.isStereotypeApplied(thisModule.HiveMapToFunctionStereotype)
  )
 to
  t : XTEXT!Transition (
   kind <- s.kind,
   source <- s.source,
   target <- s.target,
   name <- s.name,
   extension_HiveBaseMapToBehavior <- t1
  ),
  t1 : XTEXT!HiveMapToFunction (
   threadId <- s.getValue(thisModule.HiveMapToFunctionStereotype, '
    threadId'),
   newTask <- s.getValue(thisModule.HiveMapToFunctionStereotype, '
    newTask'),
   taskPriority <- s.getValue(thisModule.HiveMapToFunctionStereotype,
    'taskPriority'),
   actionPackageFile <- s.getValue(thisModule.
    HiveMapToFunctionStereotype, 'actionPackageFile'),
   actionPackageName <- s.getValue(thisModule.
    HiveMapToFunctionStereotype, 'actionPackageName')
  )
}
```

Listing 5.7: An ATL matched rule generated from a stereotpe

  To be able to obtain such a rule, HOT rule needs to be as follows:

  From the Hive profile, each stereotype that is not abstract is transformed to one ATL matched rule. Part of the ATL HOT rule to generate this is given in listing 5.8 below. In the listing, line 3 fetches all the stereotypes from the Hive profile and checks if this stereotype is not abstract. If the stereotype is not abstract then an ATL matched rule (like the one in listing 5.7) will be created with the name of the stereotype and the word "StereotypesClass". Note that the name of the rule can be any name the user wants. In listing 5.6 the stereotype name is HiveMapToFunction and therefore the name of the rule becomes HiveMapToFunctionStrereotypedClass. The generated ATL matched rule will also have an inPattern and OutPattern (see line 13 and 14 of listing 5.8). The inPattern contains the input element and a filter. The filter is needed here so as to check if a stereotype is applied to an instance of the class being transformed. Line 23 to 49 of listing 5.8 defines this filter and an example of the output(a filter) from this line is shown on line 5 of listing 5.7. This filter checks if the stereotype HiveMapToFunction is applied to an instance of a Transition.

The input element is of UML model type and is represented by a variable named 's' (see line 51 to 63 of listing 5.8). This input pattern is seen on line 3 of listing 5.7.

The outPattern has an out element of type Xtext model and is represented by a variable named 't' (see line 65 to 82 of listing 5.8). This output corresponds to the class that the stereotype extends. An example of this output pattern can be seen on line 8 of listing 5.7 and in this case the stereotype extends the Transition UML metaclass that is why the output pattern is of type XTEXT!Transition.

Another out element is created in line 94 of listing 5.8, which will represent the output from the stereotype. This out pattern is of type Xtext model and is represented by a variable named 't1'. An example of this output pattern can be seen on line 15 of listing 5.7.

Bindings for the first out element are added in line 87 and 91 of listing 5.8. An example of these bindings is shown from line 9 to 13 of listing 5.7. Bindings to the second out element created in line 93 to 96 of listing 5.8 and an example of output from these lines is shown in line 16 to 20 of listing 5.7. These bindings comes from the stereotype tagged values, and in this case our stereotype which is HiveMapToFunction has 5 tagged values which are threadId, newTask, TaskPriority, actionPackageFile and ActionPackageName.

```
1  rule fromStereotypes {
2
3    from s:PROFILE!Stereotype (not s.isAbstract)
4
5    using {
6
7      pat : ATL!SimpleOutPatternElement = OclUndefined;
8    }
9
10   to t:ATL!MatchedRule (
11
```

```
    name <- s.name + 'StereotypedClass',
    inPattern <- inpat,
    outPattern <- outpat

  ),

  inpat:ATL!InPattern (
    filter <- fil,
    elements <- elementIn
  ),

  fil:ATL!OperationCallExp(
    operationName <- 'isStereotypeApplied',
    source <- ss1,
    arguments <- ss2
  ),

  ss1: ATL!VariableExp (
    referredVariable <- varDecl
  ),

  varDecl : ATL!VariableDeclaration (
      id <- 's',
      varName <- 's'
  ),

  ss2:ATL!NavigationOrAttributeCallExp(
    name <- s.name + 'Stereotype',
    source <- thisModuleVar
  ),

  thisModuleVar:ATL!VariableExp (
    referredVariable <- thisModuleDeclaration
  ),

  thisModuleDeclaration : ATL!VariableDeclaration (
      varName <- 'thisModule'
  ),

  elementIn :ATL!SimpleInPatternElement (
        varName <- 's',
        type <- t1
  ),

  t1 : ATL!OclModelElement(
      name <-  s.getAllExtendedMetaclasses().first().name,
      model <- m
  ),

  m : ATL!OclModel (
      name <- 'UML'
  ),

  outpat:ATL!OutPattern (
    elements <- elementOut
  ),

  elementOut:ATL!SimpleOutPatternElement (
    varName <- 't',
    type <-  t2--,

  ),

  t2 : ATL!OclModelElement(
      name <-  s.getAllExtendedMetaclasses().first().name,
      model <- m2
  ),

  m2 : ATL!OclModel (
      name <- 'XTEXT'
      ),


do{

  for (e in s.getAllExtendedMetaclasses().first().getAllAttributes()) {
       elementOut.bindings <- thisModule.createBindingsFromProperties(e.
    name); -- Binding from inherited classes

      }
      elementOut.bindings <- thisModule.createExtensionBindings(s); --
    extension reference to stereotype
      outpat.elements <- thisModule.outPatternForNestedRule(s, t); --
    second output element
  for (e in s.getAllAttributes() ->select (e| not (s.
    getAllExtendedMetaclasses()->collect(a|a.name) ->includes(e.type.name)
    )) ){ -- Bindings for t1 out pattern
        pat <- t.outPattern.elements -> select(e|e.varName = 't1').first();
        pat.bindings <- thisModule.createNewBindingLink(e.name, e.name,s);
      }


    }
```

Listing 5.8: A HOT rule to generate ATL matched rules from stereotypes

- Generating ATL Helpers

  Each stereotype that is not abstract is transformed to a helper that identifies the stereotype by name from the Hive Profile in the transformation. (an example of a helper function generated is shown in listing 5.9 and this helper function is called in the generated matched rule in line 5 of listing 5.8).

```
helper def: HiveMapToFunctionStereotype : PROFILE!Stereotype =
  PROFILE!Profile.allInstancesFrom('IN1') ->select(p | p.name='HiveProfile')
  ->first().ownedStereotype->select(s | s.name='HiveMapToFunction')->first();
```

Listing 5.9: ATL Helper generated from a Stereotype.

Some helpers can be very complex and it is therefore better to write them manually and place them in a separate files. They can be called from the transformations and even reused in multiple transformations.

When deciding whether to generate these instance model transformations or write them manually, one should consider the percentage of one to one mappings in the DSL, the more one to one mappings the more value you get from generating the transformations. However if the DSL considered has very few one to one mappings, it is better to manually write the transformations rather than writing a transform to generate them.

The complete ATL code for the HOT transformation that generates a transformation from UML instance model to Hive text instance model is shown in appendix A of this report.

5.1.5 Case 2

The second case of the research was to apply the alternative one solution to the Hive Structure DSL. The Hive Structure DSL consists of a Hive Structure Profile that is used to model how the different parts of a system fit together. The hive Structure profile contains 2 stereotypes.

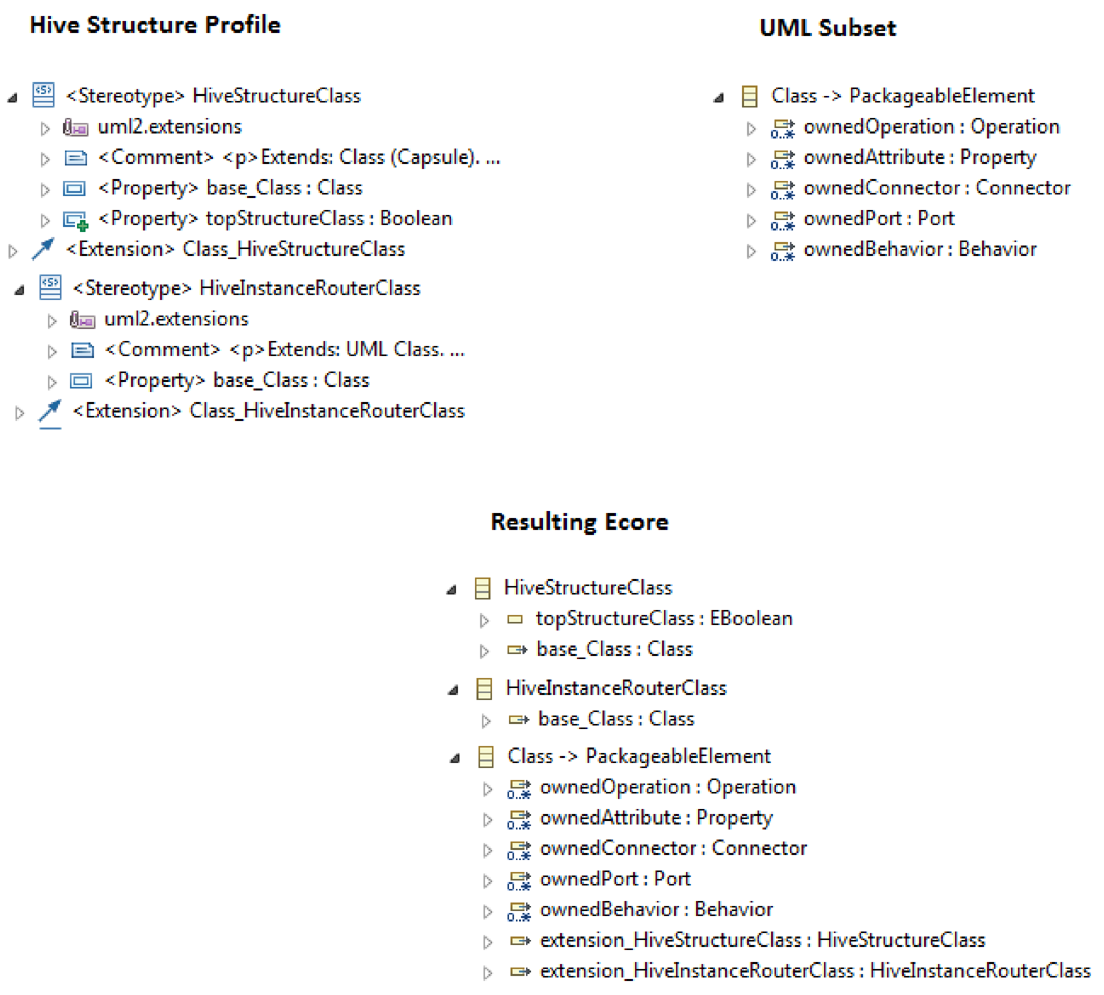
5.1.5.1 Metamodel Transformation

To obtain a metamodel for the textual notation first a subset of the UML classes used with the structural part had to be obtained. This was done by manually going through Hive instance models and identifying the classes and attributes used. With a list of these classes and attributes an Ecore of the UML subset was created. This Ecore model of the

UML subset contained 34 model elements which is 12.8% of the entire UML metamodel. A transformation that takes the Hive Structure Profile and the ecore model of the UML subset to produce an Ecore model for use in Xtext was written. The mappings used in the transformation are as described in chapter 5.1.1.

An example of part of the Hive structure Profile, UML subset and the generated Ecore for Xtext is shown in figure 5.12 below. In the figure a stereotype named HiveStructure-Class is transformed to a class called HiveStructureClass in the resulting Ecore model. The tagged value from the stereotype which is called topStructureClass is transformed to an attribute called topStructureClass in the HiveStructureClass class in the resulting Ecore model. The base_Class reference is transformed to a reference in the resulting Ecore as well. The class from the UML subset is copied to the resulting Ecore and a reference to the HivestructureClass class is added since the HivestructureClass stereotype extends the Class metaclass from UML metamodel. This reference is called 'extension _HiveStructureClass'. This same procedure is done for the HiveInstanceR-outerClass stereotype by the transformation to produce the HiveInstanceRouterClass class, base_Class reference in the HiveInstanceRouterClass class and a reference called extension_HiveInstanceRouterClass in Class.

The main difference between the first case and the second case is that in the profile of the second case there is a stereotype that extends more that one UML metaclass. This was not present in the first case.



Hive Structure Profile

```
<Stereotype> HiveStructureClass
    uml2.extensions
    <Comment> <p>Extends: Class (Capsule). ...
    <Property> base_Class : Class
    <Property> topStructureClass : Boolean
<Extension> Class_HiveStructureClass
<Stereotype> HiveInstanceRouterClass
    uml2.extensions
    <Comment> <p>Extends: UML Class. ...
    <Property> base_Class : Class
<Extension> Class_HiveInstanceRouterClass
```

UML Subset

```
Class -> PackageableElement
    ownedOperation : Operation
    ownedAttribute : Property
    ownedConnector : Connector
    ownedPort : Port
    ownedBehavior : Behavior
```

Resulting Ecore

```
HiveStructureClass
    topStructureClass : EBoolean
    base_Class : Class
HiveInstanceRouterClass
    base_Class : Class
Class -> PackageableElement
    ownedOperation : Operation
    ownedAttribute : Property
    ownedConnector : Connector
    ownedPort : Port
    ownedBehavior : Behavior
    extension_HiveStructureClass : HiveStructureClass
    extension_HiveInstanceRouterClass : HiveInstanceRouterClass
```

FIGURE 5.9: Obtaining an Ecore model for use in Xtext.

The obtained Ecore model was used as input to Xtext to generate the grammar for the Hive Structure part of the DSL. The generated grammar was then edited to our liking.

5.1.5.2 Instance Model Transformation

With the editor, it was now possible to create models in text. The next step was to create transformations that facilitate the switch from textual notation to graphical notation and vice versa. This transformations were written using ATL and the mappings adhere to those described in chapter 5.1.1.

5.1.6 Results and Analysis

With the ability to switch from UML to text and vice versa, what followed was to analyze if there is any information that gets lost or is added during the switch. To achieve this, we used demo models that are available at Ericsson which are created using the Hive graphical DSL, then switch to Xtext and back to graphical format again. The original UML file was compared with the one generated from Xtext using EMF compare. EMF compare is an Eclipse plugin that is used for comparison and merging of any EMF model [29]. Another comparison was made when switching from instance models created using the Xtext editor to UML and again back to Xtext. Since there were no models available in text, new ones were created for the purpose of this comparison. The text models were obtained from the three test UML demo models by transforming and serializing them as text. The comparison was made using the textual quickdiff functionality in Eclipse. This is a functionality that lets a user compare text files side by side. The results are given below. Note that the results for this alternative in case 1 and case 2 are similar and all are discussed here.

5.1.6.1 Graphical to Text and back to Graphical instance model

Three demo models were used for this comparison. First model is called ActivatorDemo, this demo model was initially created to demonstrate the Behavior part of the Hive profile. The second demo model used is called CBB Feature demo model which is a demo model that was created to illustrate the use of Hive Behavior and Structure. The third demo model is called Actor Demo and this was created to illustrate the use of the Hive structure profile. The model elements and the quantity present in the models are shown in the table below. Note that «Hive Activator»Class means a UML class with Hive Activator stereotype applied.

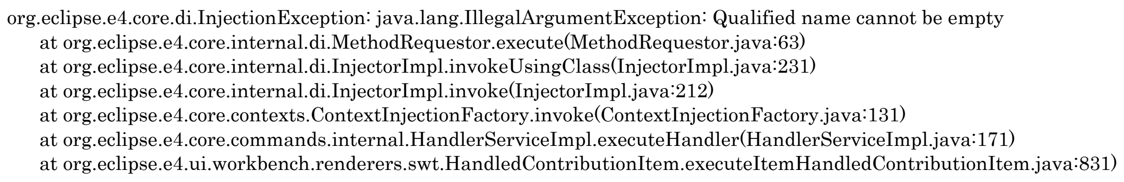
Model Element	ActivatorDemo	CBBFeatureDemo	ActorDemo
Model	✓	✓	✓
Package	✓	✓	✓
«Hive Activator» Class	✓	✓	✗
«Capsule, Hive Actor» Class	✗	✓	✓
«Hive Reactor» Class	✗	✗	✗
«Hive StructureClass» Class	✓	✓	✓
«Hive Instance RouterClass» Class	✓	✓	✗
«Hive Action» Call Operation Action	✓	✓	✗
«Hive Vector Action» Call Operation Action	✗	✓	✗
«Hive HiveInlineActivity» Call Behavior Action	✗	✗	✗
«Hive HWA Action» Call Operation Action	✗	✗	✗
«RTConnector»Connector	✗	✓	✓
«Hive Condition»Control Flow	✓	✓	✗
«Hive Map To Function»Transition	✗	✓	✓
«Hive Map Unkown signal To Function»Transition	✗	✓	✓
«Hive Map To Activity»Operation	✓	✓	✗
«Hive Go To»Send Signal Action	✗	✗	✗
«RTPort» Port	✗	✓	✓
Class	✓	✓	✗
Operation	✓	✓	✓
Property	✓	✓	✓
Statemachine	✗	✓	✓
State	✗	✓	✓
Region	✗	✓	✓
Transition	✗	✓	✓
Pseudostate	✗	✓	✓
Control Flow	✓	✓	✗
ActivityFinalNode	✓	✓	✗
Initial Node	✓	✓	✗
Accept Event Action	✓	✓	✗
Join Node	✗	✓	✗
Decision Node	✗	✓	✗
Fork Node	✗	✓	✗
Merge Node	✗	✓	✗
Instance Specification	✓	✗	✗
Dependency	✓	✗	✗
ConnectionPointRerefence	✗	✓	✗
Trigger	✗	✓	✓
Event	✗	✓	✗
ConnectorEnd	✗	✓	✓
Comment	✗	✓	✗
CallEvent	✗	✓	✓
ProtocolConformance	✗	✓	✓
SendSignalAction	✗	✗	✗
CallBehaviourAction	✗	✗	✗
Percentage DSL Coverage	39%	83%	41%

TABLE 5.2: Demo model contents

As seen from the table most of the model elements (89%) were tested with the transformations using demo models. The elements that were not tested are those that are present in the Hive DSL but have not been used in any demo models. These were tested by creating dummy elements to verify that the transformation works. From the above models the results obtained are discussed below:

Names in all model elements

One major problem was encountered when switching between the textual and graphical views. This problem is the mechanism of identifying model elements that is used in Xtext versus that used in UML. Xtext uses Qualified names to identify elements in a model and uses these same names when making references. On the contrary in UML names of model elements are optional and UML gives all model elements some unique IDs that are used to identify them. So mostly when designers model using UML they do not give names to all the model elements they create. This was the case for all the three demo models used for testing. None of them had names in all the models elements, only some model elements were named. Transformation from such UML models to Xtext produced Xtext model elements that are also unnamed. This is a problem as in Xtext we cannot have a reference to an element with no name. So the resulting model breaks.



```
org.eclipse.e4.core.di.InjectionException: java.lang.IllegalArgumentException: Qualified name cannot be empty
    at org.eclipse.e4.core.internal.di.MethodRequestor.execute(MethodRequestor.java:63)
    at org.eclipse.e4.core.internal.di.InjectorImpl.invokeUsingClass(InjectorImpl.java:231)
    at org.eclipse.e4.core.internal.di.InjectorImpl.invoke(InjectorImpl.java:212)
    at org.eclipse.e4.core.contexts.ContextInjectionFactory.invoke(ContextInjectionFactory.java:131)
    at org.eclipse.e4.core.commands.internal.HandlerServiceImpl.executeHandler(HandlerServiceImpl.java:171)
    at org.eclipse.e4.ui.workbench.renderers.swt.HandledContributionItem.executeItemHandledContributionItem.java:831)
```

FIGURE 5.10: Errors due to missing names in the model.

To solve this problem, we generate default names for elements that are not named in UML but require names in Xtext. These names are generated by the use of ATL helpers in our ATL transformations. Listing 5.8 below shows an example of a helper that generate names for Initial nodes in activity diagrams. Generating these names means that all the model elements produced by a transformation from UML to Xtext will have names. But this also means that when transforming back to UML, the elements will also be named even though they did not have names at the beginning. So the designer may notice a difference in the model due to added names.

```
helper def: getInitialNodeName(node:MM! InitialNode): String =
  if (node.name.oclIsUndefined()) then 'initialNode'+ node.activity.name
  else if (node.name.size()=0) then 'initialNode' + node.activity.name
  else node.name
  endif endif;
```

LISTING 5.10: ATL Helper to generate default names for Initial Nodes in Activity diagrams

However, if a UML instance model has names for all its model elements, then the Xtext instance model will take the same names as well and transforming back to UML will result to a model identical with the original UML instance model.

Graphical layout of the model
When a user creates models in UML, they can arrange/ format the diagram in a way that is suitable for them. For instance they can make the icons bigger, move the icons to certain positions or arrange the icons in a particular way. When these models are transformed to Xtext model and then back to UML model, the diagrams are lost since the transformation only transforms the semantic model and not any other accompanying information. In order to get the diagrams back the user has to re-generate them. This means that all the customization made in the diagrams in the original model will be lost and the user will have to redo them.

Inconsistent Models
An inconsistent model is model that does not adhere to the constraints of the metamodel i.e. models that have errors. There are scenarios when a designer may want to switch from one view to another with an inconsistent model. This scenario led to the following discovery.

Transforming an inconsistent UML model will also lead to an Xtext model that is inconsistent. This inconsistent model cannot be serialized in the Xtext syntax. This is because serializing an inconsistent model to the Xtext grammar format leads to an empty file and errors in the console showing the inconsistencies in the model with reference to the Xtext metamodel. This is not a bug in the transformation but the rather the way Xtext serialization works. The designer therefore needs to make sure that when transforming a model from UML, this model should not lead to an inconsistent model in Xtext. For example in the Hive case the join node in Xtext metamodel has a constraint that it can have several inputs and only one output. So if a designer tries to transform a join node that is not connected to any input or any output then the result of this transformation cannot be serialized in Xtext textual syntax. Instead the designer will get an error message saying that a join node must have at least 2 inputs and at most one output. The error must be fixed first for the serialization to work.

5.1.6.2 Text to Graphical and back to Textual Instance Model

Comments from Xtext
If an instance model in Xtext had comments in it, and this instance model was transformed to a UML instance model and then back to Xtext instance model the comments

were lost. This is because the XText parser ignored all the comments when creating an Xtext model. So the comments are only present in the editor but not in the resulting semantic model. If one wants the comments to be part of the model then the metamodel needs to contain an element that will hold the comments and not use the original Xtext comments available.

When an Xtext instance model (with no comments) was transformed to a UML instance model and then back to an Xtext instance model. The original model was compared with the new model using the Xtext textual diff functionality and there was no difference in the models. The models were completely identical.

Textual layout (Pretty Printing)
When a designer styles his or her text in a certain format using the text editor and then transforms the file to UML and then back to text again, this formatting is lost. This is because the transformation generates a completely different file with the default style template set in Xtext.

Inconsistent Models
When transforming an inconsistent Xtext instance model to UML, the transformation will also produce an inconsistent UML instance model with the same errors present. However when transforming this new inconsistent UML instance model back to Xtext model, the serialization to Xtext grammar fails and gives an error regarding the inconsistency in the model.

5.1.7 Discussion

This alternative proves to be a good alternative for situations where the graphical DSL has been defined using UML and UML Profiles. The fact that the textual metamodel can be derived from the Profiles means that there is no need to redefine the metamodel. In case the DSL evolves changes can be made in the Profiles and the textual metamodel can be derived again. With this solution however, one has to create and maintain a transformation that will transform the UML Profiles into an Ecore model that can be used for Xtext. As already mentioned earlier also, UML is very huge, using the whole metamodel to generate the Ecore will lead to a very huge textual grammar as well. Using a subset of the UML metamodel is therefore a preferred solution. Nevertheless, there are situations where it is not known which parts of UML are actually used and this can make getting a complete subset trickier.

Instance model transformations that transform UML models to Xtext models and vice versa also need to be created and maintained as well. If these transformations are generated from Higher Order Transformations, the Higher Order Transformations need to be maintained as well in case the DSL evolves.

Currently a major drawback to this solution is the inability to maintain customized graphical layouts. If a user had made some customization on the layout of the diagrams in UML models and switched to textual format and then back to UML again, all the diagrams will be lost in the new UML model. This means that the user needs to regenerate them and redo the customization or keep the default auto layout of the generated diagrams.

5.2 Alternative 2

The second alternative assumes that a DSL does not use UML at all. This alternative was also investigated so as to give the company the ability to explore other solutions if they wanted to abandon the existing UML based solution. This alternative describes the possibility of having a DSL supporting both graphical and textual views when the metamodel for the DSL is defined using Ecore. The major difference with this alternative compared to the first one is that with UML, graphical editors are already available in existing off-the-shelf UML tools, but defining a DSL in Ecore means that one needs to also create both the graphical editor and textual editor. There are several tools / plugins built based on EMF that can be used to create graphical editors. GMF has been selected to illustrate how this solution works because there are already some initiatives in the eclipse community on the integration of Xtext and GMF[30].

This alternative assumes a case where one metamodel is suitable for the derivation of both graphical and textual views. In these cases one metamodel which is in Ecore can be used to create both textual and graphical editors. To create the textual editor, Xtext has been used and to create the graphical editor, GMF has been used.

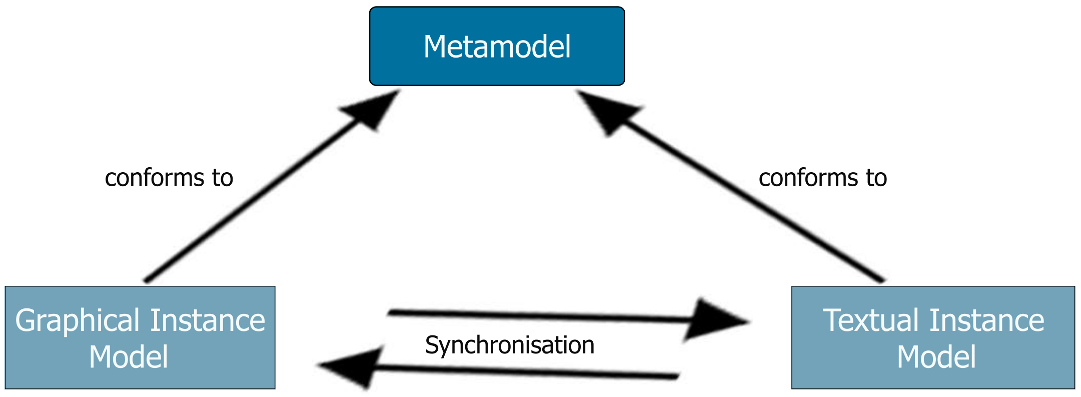


FIGURE 5.11: Alternative two.

Creating the Textual Editor

Like in the first alternative, Xtext is used as the tool to create the textual concrete syntax. In this case the metamodel which is in Ecore is supplied to Xtext and Xtext will generate the grammar. In case the generated grammar does not satisfy the user then it can be edited to fit the users' requirements. Afterwards a textual editor and parser is generated. This is automatically done with Xtext.

Creating the Graphical Editor

The same metamodel that was used to generate grammar for Xtext is now used to generate the graphical editor. With GMF we can define which figures, symbols and shapes we want to use for the language. Once the editor is complete then users can use it to create models in graphical format. The next step needed is to be able to switch between the graphical and textual views. This is described below.

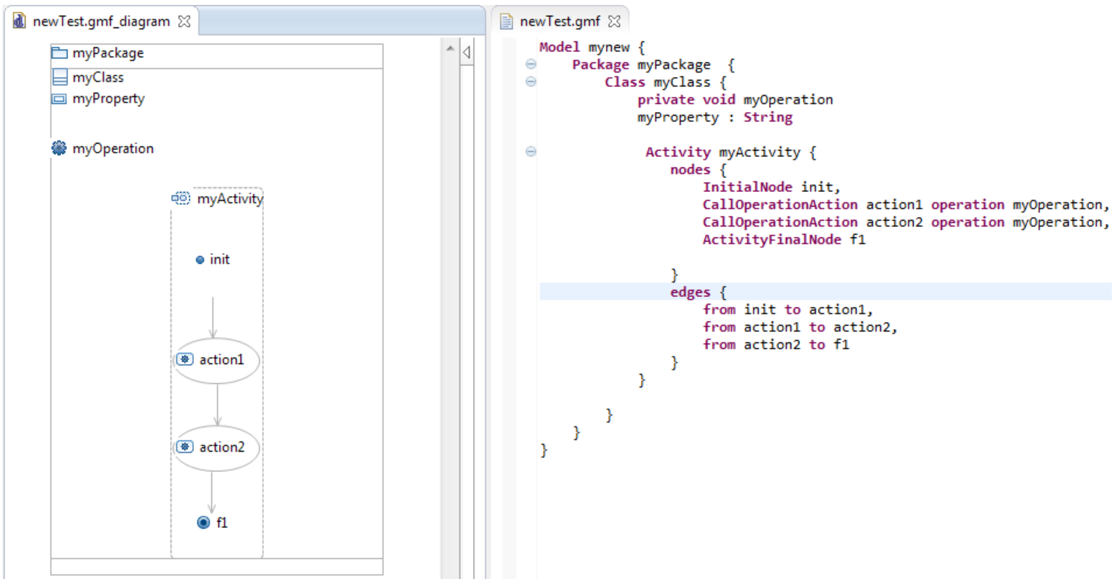
Link Textual and Graphical Editor

Since the two editors use the same metamodel then there is no need for any transformations to be done to enable switching between views. What is needed instead is a way to keep the textual and graphical files in sync so that when one is updated, the other one can receive the changes as well. GMF usually contains two types of resources, these are the abstract syntax model and the diagram model. The diagram model contains information on the notation of the model i.e shapes, font size e.t.c. To make GMF work with Xtext, we modify GMF so that it writes its abstract syntax model in text instead of XMI as usual. To avoid any clashes in the editors each editor should use a separate memory instance of the model[30] and the two will then be synchronized on save. To facilitate updating of the models, GMF uses a listener that listens to the semantic parts of the models and then updates the diagram whenever these semantic parts of the model change. In Xtext there is a function known as 'XtextResource.update' which triggers update of a model. Some glue code also needs to be written to take care of concurrency issues when the same element has been edited using both editors before saving [30].

5.2.1 Prototype

To get a good feeling on how this alternative would work a prototype was created. Due to time constraints of the thesis a complete solution for the Hive DSL was not created but rather a very small prototype was created. The prototype reused the initial textual language that was created in the first alternative. In addition a graphical editor was created using GMF that would facilitate modeling a class containing properties, operations and activities.

The generic GMF and Xtext glue code provided in [30] was used to sync the two editors. The figure below shows the results of the two editors used side by side.



```
Model mynew {
    Package myPackage  {
        Class myClass {
            private void myOperation
            myProperty : String

             Activity myActivity {
                nodes {
                    InitialNode init,
                    CallOperationAction action1 operation myOperation,
                    CallOperationAction action2 operation myOperation,
                    ActivityFinalNode f1

                }
                edges {
                    from init to action1,
                    from action1 to action2,
                    from action2 to f1
                }
            }

        }
    }
}
```

FIGURE 5.12: GMF and Xtext editors Integrated.

5.2.2 Discussion

While this alternative proved to be very flexible since it does not have any constraints brought about when using UML it also proved to require a lot of time to implement. To be able to get a decent graphical editor that can be used, one needs to invest a lot of time in creating this editor. This is due to the fact that creating graphical editors using GMF is a complex task and has a steep learning curve [31]. It may be easy to get the functionality of adding shapes and connections in the editor but there are a lot more implicit requirements of an editor that one needs to consider. For instance users expect editors to have drag and drop functionality, copy paste, select-all and many others. This requires time and certain level of knowledge to accomplish.

The advantages of using only one metamodel for the graphical and textual editor are first in case the DSL evolves, changes only need to be added to this one metamodel. The graphical editor and textual editor can then be updated as well to match the evolution of the DSL. Second there is no need to write any transformations to be able to use the graphical and textual editors in sync. This means that there will be no effort required to write or maintain any model transformations. However using only one metamodel also reduces the flexibility of the DSL. This is because all information in the graphical editor needs to be present in the textual editor as well. In many cases the graphical representation may require information that is irrelevant in the textual representation. A good example of this is the representation of control flows for activity diagrams, in text it may be enough to write that one node depends on the other but in graphics there is actually a need to create the control flows. So when deciding on whether to use one metamodel or not one needs to think about this trade-off.

5.3 Alternative 3

This alternative proposes the introduction of a core metamodel, and from this metamodel both metamodels for the graphical and textual notations can be derived. This means that in the core metamodel all information needed for both the textual and graphical notation is defined. Model to model transformations can then be used to obtain the graphical metamodel and the textual metamodel. There was no prototype implemented for this alternative due to time constraints, therefore the discussion made below is based on theoretical knowledge and knowledge gained from implementing prototypes for alternative one and two.

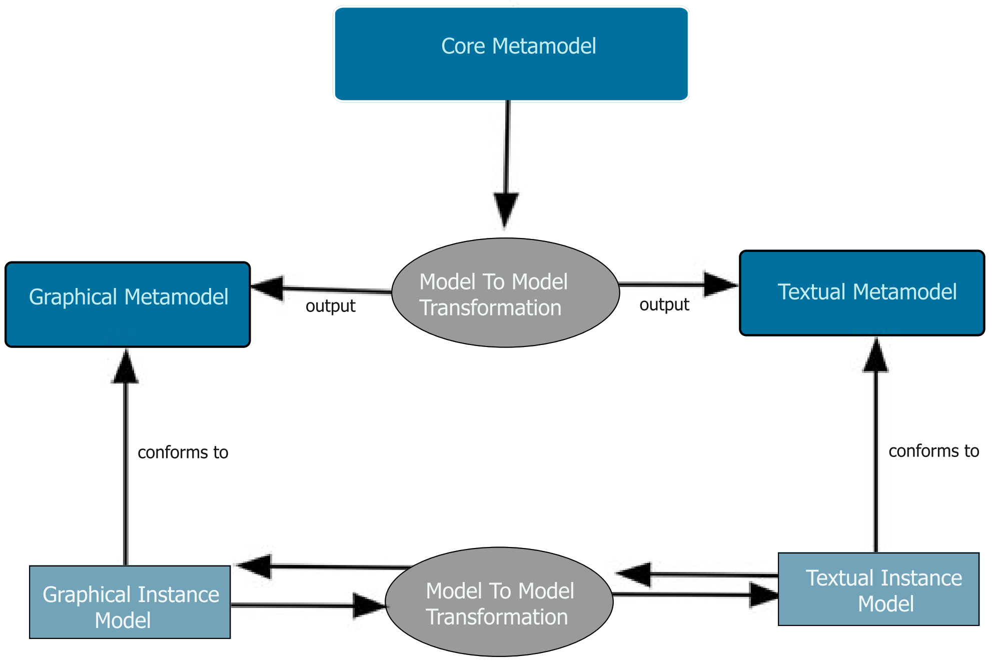


FIGURE 5.13: Alternative three.

Core Metamodel The core metamodel is created using Ecore and within it information needed for both the graphical and textual notation of the DSL is included. Then a model to model transformation is written that takes this core metamodel as input and produces two metamodels, one for the graphical notation and another one for the textual notation. The produces metamodels are then used to create editors for the DSL.

Graphical Metamodel and Editor
The graphical metamodel is used in GMF to create the graphical editor for the DSL.

Textual Metamodel and Editor
The textual metamodel is used as input to Xtext to generate the grammar and editor for the DSL

Switching Between Views
To be able to switch between the graphical and textual notation, model to model transformations are required. One transformation to transform the textual instance model to graphical instance model and one transformation to do the reverse. If the relationship between the textual and graphical metamodel is mostly one to one, these transformations can be generated.

5.3.1 Discussion

As in alternative 2, this alternative proved to be very flexible since it does not have any constraints brought about when using UML it also proved to require a lot of time to implement. To be able to get a decent graphical editor that can be used, one needs to invest a lot of time in creating this editor. It may be easy to get the functionality of adding shapes and connections in the editor but there are a lot more implicit requirements of an editor that one needs to consider. For instance users expect editors to have drag and drop functionality, copy, paste, select all and many others. This requires time and certain level of knowledge to accomplish.

Using three metamodels on the other hand provides the advantage of flexibility to the DSL since the graphical and textual metamodels are separate. However this solution adds the overhead of writing and maintaining transformations that will generate the graphical and textual metamodel from the core metamodel. There is also a need to have two instance model transformations that transform the graphical model to textual model and vice-versa. Even though there is a possibility to generate these transformations via Higher Order Transformations, these Higher Order Transformations still need to be maintained as well.

Chapter 6

Threats to Validity

This section discusses the theeats that may affect the validity of this research. The threats are grouped according to [32]

6.1 Construct Validity

This validity threat is concerned with the extent to which the studied operational measures reflect what the researcher intends to investigate. To minimise this threat the definition of the problem, research questions and hypotheses of the study was done in several iterations. These iterations included discussions with the software engineers who had the problem at Ericsson, the university supervisor who has experience in model driven engineering and the researcher. The researcher also went back to these defined research questions and hypotheses to make sure that the intended goal of the study was not lost on the way.

6.2 Internal Validity

Threats to internal validity occurs when there other external factors that affect the factor being investigated. For this research conducted there were no external factors identified that could affect the outcome of the study. However, the literature review of this research was conducted using convinient sampling when selecting the papers. Convinient sampling is the kind of sampling where candidates/artifacts are selected due to their availability and accessibility. The research papers where found from search results from google scholar website. Although the domain of multiple notations for the same DSL does not have a lot of literature there is still a slight chance that some important literature that colud reveal some external factors to the study might have been missed.

6.3 External Validity

External validity describes if the results obtained from the study can be generalized to other similar cases. To minimise threats to this validity, alternative one was tested on two cases of the Ericsson's DSL and based on the fact that the Ericsson DSL uses a profile that is standard (It is defined in a general way that any other profile can be defined), the results of this alternative can be generalized to other cases that use UML and UML Profiles for their DSL. With alternative two, a simple prototype that was very general was implemented and thus the results are transferable to other cases where the same technologies are used. With alternative three there was no implemented case so the generalizability of the results is not certain.

6.4 Reliability Validity

This validity aspect is concerned with to what extent the results of the reseach are dependent on the researcher. This poses the question, would the results change if a different researcher conducted the same research? Since the reasearch was conducted at Ericsson, the first alternative investigated was mainly due to the existing tooling used at the company. Also a lot of time was spent implementing this alternative and coming up with full prototype. On the other hand, alternative two was analysed based on a simple small prototype and alternative three was analysed based on theory. This may affect the validity of the conclusions that tend to favour alternative one over the others.

Chapter 7

Conclusions and Future Work

This chapter gives a summary of the conclusions and findings of the entire thesis work. It starts with an overall description of the thesis work, its aim, work done and the results obtained from the work done.

7.1 Summary

The aim of this thesis work was to investigate the possibility of having a DSL that supports both textual and graphical notation. The work aimed to investigate an efficient and effective way of having both graphical and textual notation in use at a company. The concerns addressed were how much maintenance effort is added to maintain the two notations, is there any loss of information when switching from one view to another and how to deal with inconsistent models when switching between the two notations.

7.2 Conclusions

To avoid doubling the maintenance effort of the two notations it is best if the DSL can have one single point of update. This was investigated and there were three alternatives proposed. The first was to use the existing graphical metamodel at the company and from this, derive a metamodel for the textual language. This proved to be a good solution as updating of the DSL needs to be done on the graphical metamodel only, and the textual metamodel can be derived. However there is still a need to update the grammar of the language as well, if one does not wish to use the Xtext generated grammar.

The second alternative was to investigate a solution based on EMF only and not UML. This solution is also based on the assumption that only one metamodel is suitable for both the graphical and textual notation. This way the Xtext textual editor can be integrated with a GMF graphical editor and enable propagation of changes from textual to graphical instance model and vice versa. This is possible as the two editors use the same metamodel and the two editors can be linked using some glue code [30] so that changes in a node in text for instance, can be propagated as change in a similar node in the graphical model. For this scenario since there is only one metamodel, updates to the DSL need to be made to only this metamodel. But the corresponding editors still need to be updated accordingly. Also there is a need to invest a considerable amount of time to create the graphical editor as it is not provided like in the UML case.

The third alternative was also a complete EMF solution based with the assumption that the textual metamodel and graphical metamodel are different. In this kind of case a core metamodel that contains both information for the graphical and textual metamodel is created. From this metamodel a model to model transformation is used to generate the textual and graphical metamodels. These metamodels are then used to generate a textual and graphical editor accordingly. This approach enables updates of the entire DSL to be done on the core metamodel since the corresponding graphical and textual metamodels can be derived from this core metamodel. However the editors for the graphical and textual views need to be updated as well to match the changes in the metamodels. The table below summarizes the basic differences of these three alternatives:

	Alternative 1	**Alternative 2**	**Alternative 3**
Number of Metamodels	2	1	3
Metamodel Transformation Needed	✓	✗	✓
Instance Model transformations needed	✓	✗	✓
Use Existing UML Graphical Editors	✓	✗	✗
Graphical Editor needs to be created	✗	✓	✓
Sycnchronization (glue) code needed	✗	✓	✗

Table 7.1: A summary of the three alternatives investigated

RQ1: What is the best way to have a DSL with both textual and graphical views without doubling the maintenance effort?

From the above discussion we can conclude that alternative one is the best way to have a DSL that supports both graphical and textual views without doubling the maintanance effort. This conclusion is drawn from the fact that with alternative one, the textual metamodel is derived from the graphical metamodel, the posibility of using a UML subset makes the resulting textual language small and transformations can be generated using Higher Order Transformations. Also alternative one comes with a great advantage because the graphical editor does not need to be created since it is available already in existing UML tools.

RQ2: How can we switch between Textual and Graphical DSL without any loss of information?

The possibility of switching between views without loss of information was another main research area for this thesis. The conclusion drawn from the work done is that there are two ways to be able to switch from one view to another. The first possibility is the use of model to model transformations. These transformations map information from a source model to a target model. Two transformations are required, one to transform a model conforming to the graphical metamodel to a model conforming to textual metamodel and another transformation for the vice versa. To avoid any loss of information, all information from one model must be mapped in the other model as well. If only part of the information is mapped then moving from one view to another and then back to the original view will lead to loss of information. When using ATL as a transformation language for these transformations, one can use Higher Order Transformations(HOT) to generate these transformations instead of writing them manually.

The second possibility to switch between textual and graphical views is to integrate the graphical and textual editors. Here both the graphical and textual editors use the same metamodel. When one is editing a model using the graphical editor, the textual editor listens to these changes and when the changes are saved the textual editor updates its model with the saved changes as well. The same is done if one is editing using the textual editor.

RQ3: How can syntax errors be handled when switching between them?

The last conclusion drawn from this thesis was on how inconsistent instance models are affected by the proposed solutions. Using the first alternative which was using UML, UML profiles and Xtext led to the conclusion that inconsistent models can be created and saved using both the graphical and textual editors. However when it comes to switching from graphical instance model to textual instance model, the inconsistent model is created in form of an XMI file but cannot be serialized in the textual syntax. Only consistent models can be serialized to textual syntax. Switching from an inconsistent textual model to graphical view produces an equally inconsistent graphical model i.e. with the same errors.

Using the second alternative, for the first scenario in which the textual and graphical editors use the same metamodel led to the following conclusion. An inconsistent model whether in graphical or textual format cannot be saved until the errors are corrected. This is due to the fact that saving one model either using the graphical or textual editor propagates the changes to the other model as well. Trying to save an inconsistent model leads to errors. Therefore only consistent models can be saved.

The conclusions discussed above where drawn from the investigation of this thesis work. Although the investigation was conducted at Ericsson these solutions and conclusions can also be applied to other cases where a DSL based on UML and UML Profiles is used. Conclusions from alternative two and three are also generalizable to situations where the DSL is an EMF based DSL.

7.3 Future Work

For future work researchers can investigate on ways to maintain the layout of graphics and format of text when using two editors. As mentioned previously this information is lost once a designer switches from one view to another and then back.

Furthermore, research can be done to investigate the use of incremental model transformations so as to increase the efficiency of the transformations. Currently the transformations investigated take a complete source model and transform it to a complete target model. This can be rather time consuming if the model being transformed is large. It is also very inefficient if the changes made to a model are small but the whole model has to be recreated by the transformation. Incremental model transformation is the kind of transformation where the transformation checks which model elements are changed and updates only those elements in the target model. The target model is not recreated but rather updated with changes from the source model. In [33] this has been investigated using a prototype for ATL. Incremental transformation can also be a way to preserve graphics in a model since the model is not recreated but just updated so the graphical information is not lost.

Another aspect that can be researched is how to store models and how to keep these models in sync when working with version control tools like Git [34]. Since some designers can decide to model using text and others decide to model using graphics, there has to be a decided way for storage of these models. Should they be stored as text or as graphics or should one store both versions of the model. The research here could also shed light on what are the challenges when storing text models and what are the challenges when storing graphical models.

In connection to storage of models, more research needs to be done on how the models written using the different notations can be merged. For instance if one designer is editing the model in text and the other one is editing the same model using the graphical format, how will they merge these changes. Will the merging be text based or model based?

Inter-model referencing is also another area that further research needs to be done. Inter-model references here means that one model has references to one or more elements which are in other models. For instance one graphical model can have references to other graphical models. When it comes to switching between views this has to be considered. Research needs to be done on whether only one model should be switched to textual views and keep its references to the graphical models or whether the referenced models will need to be converted to text models as well.

Appendix A

Higher Order Transformation to Generate a transformation from UML to Hive text instance model

```
-- @nsURI PROFILE=http://www.eclipse.org/uml2/2.0.0/UML
-- @nsURI EXTRA=http://www.eclipse.org/emf/2002/Ecore
-- @path ATL=/OneToOne/Metamodels/ATL.ecore

module HOTnew;
create OUT : ATL from IN : PROFILE, IN1 : EXTRA;

helper def: leftModel: String = '0_/http://www.eclipse.org/uml2/2.0.0/UML';

helper def: rightModel: String = '1_/http://www.eclipse.org/emf/2002/Ecore'
    ;

helper def: rightMetamodel : ATL!OclModel = OclUndefined;

helper def : id : Integer = 0;
helper def : thisMod : ATL!Module = OclUndefined;

rule createModule {

  from s:EXTRA!EPackage

  to t:ATL!Module (

    name <- 'newModule',
    inModels <- lefta,
    inModels <- leftb,
    outModels <- righta,
    elements <- EXTRA!EClass.allInstancesFrom('IN1'),
    elements <- PROFILE!Stereotype.allInstancesFrom('IN')
    ),
    lefta : ATL!OclModel (
      name <- 'IN1',
      location <- thisModule.leftModel,
      metamodel <- leftMetamodel

    ),

    leftMetamodel:ATL!OclModel (
      name <- 'UML'--,


    ),
    leftb : ATL!OclModel (
      name <- 'IN2',
      location <- thisModule.leftModel,
      metamodel <- leftMetamodelb

    ),

    leftMetamodelb:ATL!OclModel (
      name <- 'PROFILE'

    ),
    righta : ATL!OclModel (
      name <- 'rightM',
      location <- thisModule.rightModel,
      metamodel <- rightMetamodel
    ),

    rightMetamodel:ATL!OclModel (
      name <- 'XTEXT'


    )
}

rule copyClasses {

  from s:EXTRA!EClass (not s.abstract and s.name <> 'ControlFlow')

  using {

    fil:Sequence (ATL!OclExpression)= OclUndefined;

    }

  to  t:ATL!MatchedRule (
    name <- s.name + '2' + s.name,
    inPattern <- inpat,
    outPattern <- outpat
  ),

  inpat: ATL!InPattern(
    elements <- elementIn
  ),

  elementIn  :ATL!SimpleInPatternElement (
          varName <- 's',
          type <- t1
          ),

  t1  : ATL!OclModelElement(
        name <-   s.name,
        model <- m
        ),

 m  : ATL!OclModel  (
        name <-  'UML'

        ),

  outpat:ATL!OutPattern  (
    elements <- elementOut
  ),

  elementOut:ATL!SimpleOutPatternElement(
    varName <- 't',
    type <-   t2
  ),

  t2  : ATL!OclModelElement(
        name <-   s.name,
        model <- m2
        ),
 m2  : ATL!OclModel  (
        name <-  'XTEXT'
        )

        do {
          for (a in s.eAllAttributes) {

          elementOut.bindings <- thisModule.createBindingsFromProperties(a.
    name);
```

```

        }
        for (a in s.eAllReferences) {

        elementOut.bindings <- thisModule.createBindingsFromProperties(a.
    name);

        }
      }

}


rule fromStereotypes {

  from s:PROFILE!Stereotype (not s.isAbstract)

  using {

    pat : ATL!SimpleOutPatternElement = OclUndefined;
  }

  to t:ATL!MatchedRule (

    name <- s.name + 'StereotypedClass',
    inPattern <- inpat,
    outPattern <- outpat,
    isRefining <- false

  ),

  inpat:ATL!InPattern (
    filter <- fil,
    elements <- elementIn
  ),

  fil:ATL!OperationCallExp(
    operationName <- 'isStereotypeApplied',
    source <- ss1,
    arguments <- ss2
  ),

  ss1: ATL!VariableExp (
    referredVariable <- varDecl
  ),

  varDecl : ATL!VariableDeclaration (
      id <- 's',
      varName <- 's'
  ),

  ss2:ATL!NavigationOrAttributeCallExp(
    name <- s.name + 'Stereotype',
    source <- thisModuleVar
  ),

  thisModuleVar:ATL!VariableExp (
    referredVariable <- thisModuleDeclaration
  ),

  thisModuleDeclaration : ATL!VariableDeclaration (
      varName <- 'thisModule'
  ),

  elementIn :ATL!SimpleInPatternElement (
        varName <- 's',
        type <- t1
  ),

  t1 : ATL!OclModelElement(
      name <-  s.getAllExtendedMetaclasses().first().name,
      model <- m
  ),

 m : ATL!OclModel (
      name <- 'UML'
  ),

  outpat:ATL!OutPattern (
    elements <- elementOut
  ),

  elementOut:ATL!SimpleOutPatternElement (
    varName <- 't',
    type <-  t2--,

  ),

  t2 : ATL!OclModelElement(
      name <-  s.getAllExtendedMetaclasses().first().name,
      model <- m2
  ),

  m2 : ATL!OclModel (
      name <- 'XTEXT'
      ),

helperFunction : ATL!Helper (
          definition <- od,
          "module" <- ATL!Module.allInstancesFrom('OUT').first()
        ),
        od : ATL!OclFeatureDefinition (
          feature <- of
        ),
        of : ATL!Attribute (
          name <- s.name + 'Stereotype',
          type <- at,
          initExpression <- expr
        ),
        at : ATL!OclModelElement (
          name <- 'Stereotype',
          model <- m3
        ),

        m3 :ATL!OclModel (
          name <- 'PROFILE'

        ),

        expr :ATL!CollectionOperationCallExp (
          operationName <- 'first',
          source <- s1

        ),

        s1:ATL!IteratorExp (
          name <- 'select',
          source <- s2,
          body <- s14,
          iterators <- s18
        ),
        s2:ATL!NavigationOrAttributeCallExp (
          name <- 'ownedStereotype',
          source <- s3
        ),
        s3:ATL!CollectionOperationCallExp (
          operationName <- 'first',
          source <- s4
        ),
        s4:ATL!IteratorExp (
          name <- 'select',
          source <- s5,
          body <- s8,
          iterators <- s13
        ),

        s5:ATL!OperationCallExp (
          operationName <- 'allInstancesFrom',
          source <- s6,
          arguments <- s7

        ),
        s6:ATL!OclModelElement (
          name <- 'Profile',
          model <- m3

        ),

        s7: ATL!StringExp (
          stringSymbol <- 'IN2'
        ),
        s8:ATL!OperatorCallExp (
          operationName <- '=',
          source <- s9,
          arguments <- s12

        ),
        s9: ATL!NavigationOrAttributeCallExp (
          name <- 'name',
          source <- s10
        ),

        s10:ATL!VariableExp (
          referredVariable <- s11
        ),
        s11 :ATL!VariableDeclaration (
            varName <- 'e',
            id <- 'e'
        ),

        s12:ATL!StringExp (

          stringSymbol <- 'HiveProfile'
        ),

        s13:ATL!Iterator(
         varName <- 'p',
         variableExp <- s10
        ),

        s14:ATL!OperatorCallExp (
          operationName <- '=',
          source <- s15,
          arguments <- s17

        ),

        s15:ATL!NavigationOrAttributeCallExp (
            name <- 'name',
            source <- s16
        ),

        s16:ATL!VariableExp (
          referredVariable <- s11

        ),

        s17: ATL!StringExp (

          stringSymbol <- s.name
        ),
        s18:ATL!Iterator(
         varName <- 's',
         variableExp <- s16
        )

    do{


      for (e in s.getAllExtendedMetaclasses().first().getAllAttributes()) {

       elementOut.bindings <- thisModule.createBindingsFromProperties(e.
  name); -- Binding from inherited classes

      }

      elementOut.bindings <- thisModule.createExtensionBindings(s); --
  extension reference to stereotype

      outpat.elements <- thisModule.outPatternForNestedRule(s, t); -- Out
  pattern for the Stereotype - t1

      for (e in s.getAllAttributes() ->select (e| not (s.
  getAllExtendedMetaclasses()->collect(a|a.name) ->includes(e.type.name)
  )) ){ -- Bindings for t1 out pattern
        pat <- t.outPattern.elements -> select(e|e.varName = 't1').first();
        pat.bindings <- thisModule.createNewBindingLink(e.name, e.name,s);
      }


    }


}


rule outPatternForNestedRule( stereo : PROFILE!Stereotype, atl : ATL!
    MatchedRule ) {
  to
    element : ATL!SimpleOutPatternElement(
      id <- 't1',
      varName <- 't1',
      type <- variableExp
    ),
    variableExp : ATL!OclModelElement (
      name <- stereo.name,
      model <- m
    ),

    m : ATL!OclModel (
      name <- 'XTEXT'
      )

      do {

        element;
      }

}

rule createNewBindingLink (name : String, prop : String , stereo: PROFILE!
    Stereotype) {
  to
    atl : ATL!Binding (
      value <- val,
      propertyName <- name
    ),
    val:ATL!OperationCallExp (

      operationName <- 'getValue',
      source <- s1,
      arguments <- s2,
      arguments <- s3
    ),

    s2:ATL!NavigationOrAttributeCallExp (
        name <- stereo.name + 'Stereotype',
        source <- thisModuleVar
      ),
    thisModuleVar:ATL!VariableExp (
    referredVariable <- thisModuleDeclaration
    ),

    thisModuleDeclaration : ATL!VariableDeclaration (
      varName <- 'thisModule'
    ),

    s3 :ATL!StringExp (
      stringSymbol <- prop
    ),
    s1:ATL!VariableExp(
      referredVariable <- varDecl
      ),

    varDecl  : ATL!VariableDeclaration(
        id <- 's',
        varName <- 's'
      )
  do {
    atl;
  }
}


rule createExtensionBindings (stereo:PROFILE!Stereotype) {

  to t:ATL!Binding  (
    value <- val,
    propertyName <-'extension_' + stereo.name

  ),
  val:ATL!VariableExp(
      referredVariable <- varDecl
      ),
      varDecl  : ATL!VariableDeclaration(
        id <- 't1',
        varName <- 't1'
      )

      do {

        t;
      }

}
```

```
rule createBindingsFromProperties(propertyName:String) {

  to t:ATL!Binding   (
    value <- val,
    propertyName <-propertyName

  ),
  val:ATL!NavigationOrAttributeCallExp  (
        name <- propertyName,
        source <- s1
      ),
      s1:ATL!VariableExp(
      referredVariable <- varDecl
      ),
      varDecl   : ATL!VariableDeclaration(
        id <- 's',
        varName <- 's'
      )
      do {

        t;
      }

}
```

Listing A.1: ATL HOT to generate UML TO Hive text instance model transformation

Appendix B

ATL Transformation to Transform a UML Profile into an Ecore model

```
-- @nsURI PROFILE=http://www.eclipse.org/uml2/2.0.0/UML
-- @nsURI EXTRA=http://www.eclipse.org/emf/2002/Ecore
-- @nsURI MM=http://www.eclipse.org/emf/2002/Ecore

module UMLToEcore;
create OUT : MM from IN : PROFILE, IN1 : EXTRA;

-- Helpers Start Here
helper context PROFILE!Property def: getMatchingPrimitiveMClass() : MM!
    EcorePackage =
     if(self.type.name = 'String') then MM!EString
     else if(self.type.name = 'Boolean') then MM!EBoolean
     else if(self.type.name = 'Integer') then MM!EInt
     else if(self.type.name = 'Real') then MM!EDouble
     else self.type
     endif endif endif endif;


helper def : retrieveMetaclass(name: String) : MM!EClass =
  MM!EClass.allInstancesFrom('OUT')->select (e | e.name = name).first();


helper context PROFILE!Property def : isContainment : Boolean =
  self.aggregation.toString() = 'composite';

-- Helpers End Here


-- Matched Rules Start Here

```

```
rule Profile {
from s : PROFILE!Profile

to t : MM!EPackage(
name <- s.name,
nsURI <-s.name,
nsPrefix <- s.name,
eClassifiers <- s.packagedElement -> select(e|
not e.oclIsKindOf(PROFILE!Package)
and not e.oclIsKindOf(PROFILE!DirectedRelationship)
and not e.oclIsKindOf(PROFILE!Extension)),
eSubpackages <- s.nestedPackage,

-- Add extra classes to the HiveProfile Package
eClassifiers <- MM!EClass.allInstancesFrom('IN1'),

-- Add Enumerations that come from Extra model to the HiveProfile Package
eClassifiers <- MM!EEnum.allInstancesFrom('IN1')

)

}


rule ExtraClassesToEClasses {

  from s:EXTRA!EClass

  to t: MM!EClass (
    name <- s.name,
    eStructuralFeatures <- s.eStructuralFeatures,
    eSuperTypes <- s.eSuperTypes,
    abstract <- s.abstract
    )

}

rule StereotypeToEClass {

  from s : PROFILE!Stereotype

  to t : MM!EClass(
    name <- s.name,
    eSuperTypes <- s.superClass,
    eStructuralFeatures <- s.ownedAttribute,
    "abstract" <- s.isAbstract

    )

  do{

  for (e in s.getExtendedMetaclasses()){

    if (not s.superClass.first().oclIsTypeOf(PROFILE!Stereotype)) {
    thisModule.createExtension(s, MM!EClass.allInstancesFrom('OUT') ->
  select(m|m.name = e.name).first());
    thisModule.createSubClasses(e, s, t);

      }
    }
  }
}

rule DataTypeToEClass {

  from s : PROFILE!DataType (not s.oclIsTypeOf(PROFILE!Enumeration))

  to t : MM!EDataType(
    name <- s.name,
    eStructuralFeatures <- s.Attribute
    )
}

rule UMLClassToEClass{

from s : PROFILE!Class (s.oclIsTypeOf(PROFILE!Class))

to t: MM!EClass (
  name <- s.name,
  eStructuralFeatures <- s.ownedAttribute,
  abstract <- s.isAbstract
)

 }

rule ExtraAttributesToEAttributes {

  from s:EXTRA!EAttribute

  to t:MM!EAttribute (
      name <- s.name,
      eType <- s.eType
    )
}

rule ExtraReferencesToEreferences {
      from  s:EXTRA!EReference

      to  t:MM!EReference  (

        name <- s.name,
        eType <- s.eType,
        upperBound <- s.upperBound,
        lowerBound <- s.lowerBound,
        containment <- s.containment
      )
  }

-- For properties whose type is another Class
  rule PropertyToReference{

    from  s  : PROFILE!Property(
        (s.type.oclIsTypeOf(PROFILE!Class)
        or  s.type.oclIsTypeOf(PROFILE!DataType))
        and  not  s.association.oclIsTypeOf(PROFILE!Extension))

    to   t  : MM!EReference(
      name <- s.name,
      lowerBound <- s.lower,
      upperBound <- s.upper,
      containment <- s.isContainment,
      eType <- MM!EClass.allInstancesFrom('OUT')
      ->select(e | e.name=s.type.name).first(),
      eType <- MM!EClass.allInstancesFrom('OUT')
      ->select(e | e.name=s.type.name).first()
  )

  }

-- Properties with primitive types transformed to Attributes

  rule UMLAttributesToEattributes {
  from   s  :PROFILE!Property(
      s.type.oclIsTypeOf(PROFILE!PrimitiveType)

      and  not  s.association.oclIsTypeOf(PROFILE!Extension) )

  to   t  : MM!EAttribute(
      name <- s.name,
      lowerBound <- s.lower,
      upperBound <- s.upper,
      eType <- s.getMatchingPrimitiveMClass()
      )
  }

  rule EnumerationToEEnum {
    from s : PROFILE!Enumeration

    to t : MM!EEnum(
      name <- s.name,
      instanceClassName <- s.name,
      instanceTypeName <- s.name,
      eLiterals <- s.ownedLiteral
      )
  }

  rule EnumerationLiteralToEEnumLiteral {

    from s : PROFILE!EnumerationLiteral

    to t : MM!EEnumLiteral (
      name <- s.name
      )
  }

  rule EEnum_ExtraToEEnum {

    from s :EXTRA!EEnum

    to t : MM!EEnum(
      name <- s.name,
      eLiterals <- s.eLiterals
      )
  }

  rule EEnumLiteral_ExtraToEEnumLiteral {

    from s : EXTRA!EEnumLiteral

    to t : MM!EEnumLiteral (
      name <- s.name,
      value <- s.value
      )
  }

-- Matched Rules End Here

-- Called Rules Start Here ---

  rule createSubClasses(extendedMeta: PROFILE!Element, stereo: PROFILE!
     Stereotype, meta: MM!EClass) { -- Create the base Reference to Extended
      MetaClass

    to t: MM!EReference (
      name <- 'base_' + extendedMeta.name,
      upperBound <- 1,
      lowerBound <- 0,
      eType <- MM!EClass.allInstancesFrom('OUT')
      ->select(e|e.name=extendedMeta.name).first()

      )
    do {

      if (not extendedMeta.name.oclIsUndefined())
      {

      t.eOpposite <- thisModule.retrieveMetaclass(extendedMeta.name).
      eStructuralFeatures->select(e|e.name='extension_' + stereo.name)
      .first();
      }
      meta.eStructuralFeatures <- meta.eStructuralFeatures.including(t)
      .asSequence()->flatten();
      t.eOpposite.eOpposite <- t;
      t.eOpposite.eType <- meta;
    }
  }

  rule createExtension(stereo: PROFILE!Stereotype, meta: MM!EClass) { --
    Create Extension Reference to stereotype

    to t: MM!EReference (
      name <- 'extension_' + stereo.name,
      upperBound <- 1,
      lowerBound <- 0,
      containment <- true

      )
    do {
       meta.eStructuralFeatures <- meta.eStructuralFeatures.including(t)
      .asSequence()->flatten();
      }
  }

-- Called Rules End Here ---
```

Listing B.1: ATL Transformation to Transform a UML Profile into an Ecore model

Bibliography

[1] Uml diagrams overview. `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ed/UML_diagrams_overview.svg/500px-UML_diagrams_overview.svg.png`, 2014. Accesed: 2014-06-25.

[2] R. Baskerville. Investigating information systems with action research. *Communications of the AIS*, 2(3), 1999.

[3] Introducing higher-order transformations (hots). `http://modeling-languages.com/introducing-higher-order-transformations-hots`, 2010. Accessed: 2014-06-20.

[4] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *Model Driven Architecture-Foundations and Applications*, pages 18–33. Springer, 2009.

[5] J. Bezivin and O. Gerbe. Towards a precise definition of the omg/mda framework. *pp.*, pages 273–280, 2001.

[6] David S Wile. Supporting the dsl spectrum. *CIT. Journal of computing and information technology*, 9(4):263–287, 2001.

[7] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3: 39–60, 1997.

[8] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. In *4th International Workshop on Software Language Engineering*, 2007.

[9] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice. 1st ed. [San Rafael.* Morgan & Claypool, Calif.], 2012.

[10] Xavier Le Pallec and Sophie Dupuy-Chessa. Support for quality metrics in metamodelling. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, pages 23–31. ACM, 2013.

[11] Omg.org, (2014). omg uml. `http://www.omg.org/gettingstarted/what_is_uml.htm`, 2014. Accessed:2014-05-20.

[12] Lidia Fuentes-Fernåndez and Antonio Vallecillo-Moreno. An introduction to uml profiles. *UML and Model Engineering*, 2, 2004.

[13] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[14] N. Skrypuch. Eclipse modeling-emf-home. http://www.eclipse.org/modeling/emf, 2014. Accesed: 2014-06-13.

[15] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[16] Eclipse.org. Graphical modeling framework(gmf)-tooling. `http://eclipse.org/gmf-tooling`, 2014. Accesed: 2014-06-13.

[17] S. Efftinge. Xtext - language development made easy! `http://www.eclipse.org/Xtext`, 2014. Accessed: 2014-06-13.

[18] Eclipse.org. Atl. `http://www.eclipse.org/atl`, 2014. Accessed:2014-05-20.

[19] Emftext.org. Emftext. `http://www.emftext.org/index.php/EMFText`, 2014. Accessed: 2014-06-13.

[20] N. Skrypuch. Eclipse modeling-mmt-home. http://www.eclipse.org/mmt/?project=qvto, 2014. Accesed: 2014-06-13.

[21] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *Techn. Ber. tr-ri-07-284, University of Paderborn*, 2007.

[22] Dos Santos and Travassos P. G. and zelkowitz, m. (2011). *Action research can swing the balance in experimental software engineering*, 2011.

[23] C. Atkinson and R. Gerbig. Harmonizing textual and graphical visualizations of domain specific models. *pp.*, pages 32–41, 2013.

[24] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[25] F. Andres and de Lara. J. and guerra, e. (2008). In *Domain specific languages with graphical and textual views*, pages 82–97. Springer, Springer.

[26] L. and Engelen. and van den brand, m. (2010). *Integrating textual and graphical modelling languages*, 253(7):105–120.

[27] Gentleware.com. Uml-to-ecore plug-in. `http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/ecoreguide.html`, 2014. Accesed: 2014-06-30.

[28] Eclipse.org. About tcs. `http://www.eclipse.org/gmt/tcs/about.php`, 2014. Accesed: 2014-08-11.

[29] Eclipse.org. Emf compare - compare and merge your emf models. `http://www.eclipse.org/emf/compare/overview.html`, 2014. Accesed: 2014-06-30.

[30] http://www.eclipse.org. integration with emf and other emf editors. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.xtext.doc%2Fcontents%2F210-emf-integration.html`, 2014. Accesed: 2014-06-13.

[31] Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19. IEEE Computer Society, 2009.

[32] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[33] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Lazy execution of model-to-model transformations. In *Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2011.

[34] git scm.com. Git. `http://git-scm.com`, 2014. Accesed: 2014-06-25.