A Metamodel independent approach for Conflict Detection to support distributed development in MDE

By

Mostafa Pordel

mpl08001@student.mdh.se

A THESIS

Submitted in partial fulfillment of requirements for

Master Degree of Software Engineering

School of Innovation, Design and Engineering

Mälardalen University, Västerås, Sweden

September 10th, 2009

Approved by:

Supervisor: Antonio Cicchetti

 Examiner: Ivica Crnkovic
I. Abstract

The need for change of direction in Software Engineering has been suggested in several resources [Power of Models]. In recent years many organizations have focused on Model Driven Engineering (MDE) as an approach to application design and implementation. Model Driven Architecture (MDA) was introduced by Object Management Group (OMG) in 2001 in support of MDE. Models are the basic elements in MDE. The focus on MDE is on the concept of “Everything is a model”. So far several languages, tools and platforms have been created for MDE.

In particular, models can be developed in a distributed environment, thus once they merge together, conflicts and inconsistencies should be detected in a convenient way to be reconciled (automatically by software or manually by developers). This project is based on previous works that define difference and conflict metamodels of a source model. In this report, we introduce the benefits of versioning systems in MDE framework. A conflict metamodel that is generated from the input metamodel and the architecture for detecting conflicts are represented. The proposed approach is metamodel independent that means conflict metamodel is created for any input model which conforms to Kernel Meta Meta Model (KM3). The concept of used conflict model can be also changed regardless of models. The extended idea for conflicts detection, the presented architecture for model code management and the tools that can detect conflicts of concurrent models can help to improve MDE in model code management. For this report, some implementation in Eclipse platform has been rendered while some future works are suggested.
II. Acknowledgement

Preparation of this master degree report took about six months. I’m thankful to the lord for giving me the opportunity of learning and also to all those people who helped me during this period. I thank god for giving me the chance of living and learning. Here I want to thank all who helped me in this project during these months.

The first person I would like to thank is my wife Farahnaz who helped me both at work and at home. She is not only a great and calm person but also a fantastic colleague. She helped me a lot with some reviews, development in Eclipse and some installation and configuration of the software needed for implementation part of this project. Without her support this report was not possible in the specified time table.

I also want to thank my supervisor, Antonio Cicchetti who helped me whenever I needed. I could easily contact him all this time even when he was on trip. This report is actually an extension to his previous works in L’Aquila University. Antonio is a kind supervisor who helped me a lot during this project.

I have done this thesis in Mälardalen University and during my studies in this university, I had some courses with Professor Ivica Crnkovic and finally he was my examiner for this thesis. He honored me with some reviews on my work. I am grateful for all his support for me as his student.

Finally I want to express my gratitude to my friend Guido Diodato. He helped me with some basic principles of modeling development at the very beginning of this work.
Life is like a bicycle, to keep your balance you must keep on moving.

Albert Einstein
III. Contents

I. Abstract ......................................................................................................................... 2

II. Acknowledgement ......................................................................................................... 3

III. Contents .................................................................................................................... 5

IV. List of Figures ............................................................................................................ 7

1. Introduction ................................................................................................................... 9
   1.1. Outline of the thesis ........................................................................................ 9

2. Introduction to MDE .................................................................................................... 10
   2.1. Standards and technologies used in MDE ..................................................... 12
      2.1.1. UML ..................................................................................................... 12
      2.1.2. MOF and XMI ..................................................................................... 13
      2.1.3. CWM .................................................................................................. 14
      2.1.4. EMF ................................................................................................... 15
      2.1.5. Kermeta ............................................................................................... 16
      2.1.6. MOFScript .......................................................................................... 18
      2.1.7. MTF ..................................................................................................... 20
      2.1.8. ATL ..................................................................................................... 21

3. Version Management .................................................................................................. 23
   3.1. Model developing ............................................................................................ 24
   3.2. Difference Model ............................................................................................. 27
3.3. Model Code Management .................................................................................... 30
4. Difference Metamodel ............................................................................................. 32
5. Conflict Metamodel ............................................................................................... 35
6. Conflict Detection Architecture ............................................................................. 37
   6.1. Advantages .................................................................................................... 40
7. Implementation ...................................................................................................... 41
8. Conclusion and future works .................................................................................. 46
9. References ............................................................................................................ 48
IV.  List of Figures

Figure 1: MDA and Object technology basic concepts. ................................................................. 12
Figure 2: Classical Four Layer Metadata Architecture, ISO/IEC 19502:2005(E) [MOF_OMG] ......................... 14
Figure 3: Core of OMG metadata repository [CWM-1.1]. ........................................................................ 15
Figure 4: Kermeta Positioning [Kermeta]. ............................................................................................ 17
Figure 5: EMOF extension and Kermeta promotion [KerMeta]. ............................................................. 18
Figure 6: MOFScript Component is divided into service component and tool component [MOFScript]. ............... 19
Figure 7: MOFScript plug-ins [MOFScript] .......................................................................................... 19
Figure 8: Transformations file in MOFScript under Eclipse Platform [MOFScript]. ................................. 20
Figure 9: ATL operational context. M_a is a transformation file that conforms to ATL. M_a is the input model for M_a and M_b is the output of the transformation. All metamodels conform to MOF. ................................................................. 22
Figure 10: ATL lunch configuration [ATL] .............................................................................................. 23
Figure 11: Two new concurrent versions of Model A. Version 1 does not have “Server Node” and Version 2 has added “Edge 2”. ......................................................................................................................... 25
Figure 12: Process of updating new models into a model repository. This architecture is described with details by Takafumi Oda and others [version]. ......................................................................................... 26
Figure 13: Two version of a UML model [ModelDiff]. ............................................................................. 27
Figure 14: Visualization of emfdiff in the GUI of EMF Compare [ModelDiff]. .............................................. 28
Figure 15: Difference Model of two input model conforms to Difference metamodel which is created automatically from the metamodel of input model [Conflicts] .................................................................................. 29
Figure 16: Difference model of Version1 and Model A. Changes made for V1 and Server Node is deleted .......... 29
Figure 17: Difference Model of comparing Version 2 and Model A; Changes made for Version 2 and “Edge2” is added and “Server” is updated ................................................................................................. 30
Figure 18: NET metamodel diagram ........................................................................................................ 33
Figure 19: Difference Metamodel of NET metamodel; each element of NET metamodel is extended to three other elements that inherits Deleted, Added, and Changed elements. .................................................................................. 34
Figure 20: NET Conflict Metamodel; the idea is based on defining left and right patterns for elements of two input versions of the same model that results in conflicts. .......................................................... 36
Figure 21: Conflict Patterns in the language of metamodel NET Conflict. There may be many debates on conflict language and reconciliation strategy. Current conflict detecting solution is independent from conflict languages and also from metamodel of input model ................................................................. 37
Figure 22: Conflict Detection scenario: for different versions of input Model A, the conflict patterns are used to detect conflicts. These conflict patterns are generated by an expert with some elements of conflict metamodel of the Model A. In order to have conflict metamodel of Model A, input metamodel of Model A and a conflict language is merged. 38

Figure 23: Eclipse project 41

Figure 24: Conflict detection scenario; the green parts are implemented in this thesis, the blue parts have been partially implemented and the red part should be considered in future works. 47
1. Introduction

Models are the basic elements in Model Driven Engineering (MDE). The main principle in MDE is focusing on models; considering the concept of “Everything is a model”. This has been partially solved for programming languages: several languages, tools and platforms have been developed for this purpose and some of them are introduced in this report.

Model development has introduced many new research fields. Our research is focused on model version management. Models can be developed in a distributed environment, thus once they merge together, conflicts and inconsistencies should be detected in a convenient way in order to be reconciled. The emphasis here is on generating conflict Metamodel of models automatically. This conflict metamodel is used to generate conflict patterns. Conflict patterns represent cases in which conflicts occur with two concurrent versions of the same model. When model developers update model repositories, this conflict detection will be so useful and practical as it shows existing conflicts to the users, so the time for manual recognition of conflicts will be reduced. Also with some reconciliation strategies, conflicts could be resolved automatically.

The proposed solution is a metamodel independent approach. All models that conform to KM3 could be handled by this solution. KM3 is a domain specific language for defining metamodels and is available under the Eclipse platform. This project is based on previous works of A.Cicchetti [Antonio] that define difference and conflict metamodels. The aim of this thesis is to create an Eclipse plug-in to generate the conflict patterns in order to be used in conflict detector tools in model code managements.

1.1. Outline of the thesis

The thesis is structured as follows:

- Chapter 2 describes Model Driven Engineering (MDE) and Model Driven Architecture (MDA). It describes basic concepts for this thesis report. In this chapter models that are elements of MDE are presented. Standards, languages and open source tools that support MDE and MDA are discussed briefly. Some of these mentioned tools and standards are used in implementation phase of the project.
Chapter 3 discusses version management in Software Engineering and specifically in model development in MDE. Difference model is presented as the key solution to version management. At the end of this chapter we have introduced the Model Code Management for managing models and model repositories.

In chapter 4, the difference metamodel adopted to represent differences between model revisions, which is used in version management, is explained. The mentioned difference metamodel is used in this thesis to generate conflict metamodel.

Chapter 5 introduces a conflict metamodel used for defining the language of conflict models. This conflict metamodel is able to define conflict patterns by which we are able to find the conflicts in concurrent revisions of the same version.

Chapter 6 presents conflict detection architecture. It includes conflict language, input model, conflict language of the input model, conflict pattern, input difference model of two concurrent versions of the same model and the conflict result. All the processes of defining conflict are described in this chapter.

Chapter 7 gives some parts of implementation in Eclipse. ATL language is used to create transformation specifications.

Chapter 8 adds the conclusion to this thesis. Some optimization in conflict language, new model based language for defining patterns of conflicts and some other future works are suggested in this chapter.

2. Introduction to MDE

The most popular Software Engineering methods in last 30 years were based on the basic concept of "everything is an object". This basic idea was so successful in driving software engineering to simplicity and generality. It has provided a great integrity mechanism for software development.

After introducing many extensions and concepts in object oriented solutions, the need for change of direction in software engineering has been suggested in several papers. For example we can refer to the following statements from Greenfield and others [Patterns]:

“The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. However, growing pressure to reduce cost and time to market and to improve software quality may catalyze a transition to more automated
methods. We look at how the software industry may be industrialized, and we describe technologies that might be used to support this vision. We suggest that the current software development paradigm, based on object orientation, may have reached the point of exhaustion, and we propose a model for its successor."

Recently the idea of “Everything is a model” has raised great interests among software engineers and opened a realistic research agenda [Power of Models].

MDE is a shift from code-based to model-based programming. Figure 1 illustrates the difference between the idea of Object Oriented (OO) technologies and MDE. As it is shown, the basic component of MDE is a “Model” while the basic component of OO is an “Object”. In MDE, a model “conforms to” another model; while objects “inherit from” other objects. The two mentioned relations may appear to be the same but they are different. “Inheritance” relation presents an object that inherits all private, protected and public features from its parents with some accessibility that is defined for each of them while “Conforms to” relation indicates that the current model is written in the language of its metamodel. We should avoid picking the concepts of each of them for another. For instance, there is no base model in MDE. If the concepts and relations are used as defined, the two software development architectures can be complimentary for each other.

MDE is a very positive development for several reasons. It encourages efficient use of system models in software development process. The usage of models as the base elements of development has encouraged software developers to shift their traditional solutions to MDE or use it as a complimentary of their programs. Reuse of models for creating families of systems in different platforms and frameworks was so successful in last years that some modeling languages like UML (Unified Modeling Languages) that is not even executable is currently widely accepted in different projects, tools and languages. It is therefore a great interest over solutions like MDE to organize and manage enterprise architectures. Over last years, different groups and companies tried to utilize, improve and standardize MDE. Model Driven Architecture (MDA) that defines a layering architecture for MDE was introduced in 2001 by OMG group. It focuses on separating model of system to several models including Platform Independent Model (PIM) and Platform Specific Model
(PSM). This layering on models adds portability to software solutions. MDA is now supported by some automated tools and services for both defining the models and facilitating transformations between different model types [Eclipse].

![Diagram of MDA and Object technology basic concepts](image)

**Figure 1: MDA and Object technology basic concepts.**

A) An object inherits from another object.  B) MDA model conforms to another model.

MDE is a framework that integrates models and binary codes in a homogenous process by the means of models and transformations.

### 1.2. Standards and technologies used in MDE

Our solution for conflict detection is based on MDA. We use EMF and ATL for implementation under Eclipse platform. In this section we briefly introduce some standards, technologies and basic concepts that are widely used in MDA and MDE and may help readers understand the concepts in this project. Here the focus is on the Modeling integration view of presented notions.

#### 1.2.1. UML

Unified Modeling Language (UML) is a language for specifying and designing software systems. It provides constructs for software components, objects, activities, relations and etc. Now UML 2.0 is the current official version which is widely used in different industries. Four different parts of UML are as follows: 

---

~12~
• UML 2.0 Infrastructure [UMLInfra]
  It defines the core metamodel which superstructure specification is based on.

• UML 2.0 OCL
  Object Constraint Language is used for defining rules in model elements.

• UML 2.0 Diagram Interchange
  It defines how diagrams layouts in UML 2.0 are exchanged.

• UML 2.0 Superstructure Specification
  It defines semantic of diagrams and their model elements.

Models in MDA could be written in UML or any other languages. Several improvements have been done in UML 2 to make it closer and more applicable for MDE. MDA helps UML models to be executable [MDE-Limit]. Among the four parts of UML that were mentioned, the superstructure specification has been defined and completed. This part is available for download while the other three parts have not been finalized yet [MDA Spec]. Currently UML 2.0 helps modeling in different domains including real-time systems and embedded systems.

1.2.2. MOF and XMI
Meta Object Facility (MOF) provides standard modeling and constructs. It is another OMG standard for MDE and specifically MDA. The constructs are a subset of UML modeling constructs. MOF helps with defining the transformations between models [MDATech]. Originally MOF was defined as metamodel of UML. The main theme of MOF for metadata management is its extendibility. The aim of MOF is to provide a framework that all metadata can be defined within it. In order to achieve such a goal, MOF is selecting a classical popular four layer structure that provides a metadata management framework [MOF_OMG]. This architecture is popular within standard communities such as ISO and CDIF.
MOF has a meta-metamodel on top layer, which is named M3. Metamodel is defined on the second layer that is M2 and the model is located on M1 layer, while M0 represents an instance of a model. Each layer is defined by the language of its upper layer. MOF is a closed “metamodeling architecture” because it defines a M3 layer which conforms to itself and therefore it is able to define its own language. In this architecture, it may be used to include Object Oriented (OO) models or metamodels as in UML as well as non-object oriented models and metamodels such as web services for mentioned layers. Interchanges of models are done through another OMG standard which is XML Metadata Interchange (XMI). XMI is an extension of XML and is used to hold model’s information in structured text format [MDATech].

1.2.3. CWM

Common Warehouse Metamodel (CWM) is a data warehouse standard which has been defined by OMG. It supports management of the life cycle including designing, managing and building data warehouse applications [MDATech]. It has specifications for modeling metadata for relational, non-relational, multi-dimensional, and other types found in a data warehousing environment. It is a standard that specifies interfaces for interaction between warehouse and business intelligence metadata including warehouse platforms, warehouse tools and warehouse metadata repositories in large environments. CMW is based on UML,
MOF and XMI. Figure 3 shows how other OMG standards form the core of OMG metadata repository [CWM-1.1].

![Repository Common Facility diagram](image)

**Figure 3: Core of OMG metadata repository [CWM-1.1].**

As shown in Figure 3, UML defines object oriented models while it is widely supported by different tools and platforms. MOF presents an extendible framework for defining models and metamodels and also introduces some interfaces for interacting with metadata in repository. XMI allows interchange of data in stream format.

### 1.2.4. EMF

The Eclipse Modeling Framework (EMF) is one of the most popular projects within Eclipse community. It has largely adopted the MDA approach and has implemented most of MDA’s core specifications. These include Meta Object Facility (MOF); Unified Modeling Language (UML); Query View Transformation (QVT), Object Constraint Language (OCL), MOF to Text (MOF2Text), and XML Metadata Interchange (XMI). EMF is a project that aims to generate code from structured models. It is an open source that is targeting MDE. EMF provides java classes for models that are described in XMI files. It can generate Eclipse plug-ins and graphical editors for other Eclipse projects. It also synchronizes the changes in models before and after generating codes of models. EMF supports standard create, delete, update and add operations.
EMF has three main pieces:

I. EMF Core.
It includes ECORE metamodel for describing models. EMF code has runtime support for changed notification in models. It also has API for manipulating models.

II. EMF Edit
It has generic reusable classes for developing editors for EMF models. It includes an editor that shows models graphically and a command framework that supports difference classes for building models with automation on undo/redo actions.

III. EMF CodeGen
It has a GUI in which users can set different options for generating codes. The generated codes of models are joined to JDT (Java Development Tooling) of Eclipse.

In EMF the following three kinds of code generations are handled:

I. Models
It includes Java interfaces, classes, factory and packages for all classes in models. Factory is used for creating instances of model classes. Package gives some static constants and convenience methods for accessing metadata of models.

II. Adapters
Generate Item providers that are implementation of classes used in approving model classes for editing and presentation.

III. Editor
It is an Eclipse style editor that is considered as a start point for customizing models [Eclipse_EMF].

1.2.5. Kermeta
Kermeta has been developed by INRIA Triskell team. The Kermeta workbench is an Eclipse-based Object Oriented (OO) Meta programming environment dedicated to
metamodel engineering activities. It includes specification of abstract syntax, static semantics (OCL) and dynamic semantics with connection to the concrete syntax, model prototyping, meta-model prototyping, model simulation, model transformation and aspect weaving. Kermeta is built as an extension to Eclipse EMF.

Kermeta is a metamodel language which includes both description of models and the behavior of models. It is created to be a complement to Eclipse MOF and Ecore. Compared to MOF, it has additional description for Model’s behavior and is created to be the base language for Model Engineering covering action languages, metamodel languages, constraint languages and transformation languages needed for modeling.

Kermeta is model oriented and Object-Oriented while it uses an imperative language [Kermeta].

Below is the “hello world” program which shows how it combines modeling and Object Oriented programming.

```plaintext
@mainClass "helloworld::HelloworldExample"
@mainOperation "sayHello"
package helloworld;
require kermeta
using kermeta::standard
```
class HelloworldExample
{
    operation sayHello() is
    do
        stdio.writeln("Hello world, ...")
    end
}

Code 1: The "Hello world" example of Kermeta [Kermeta]. A class is defined in a modeling code. In most of other modeling languages like ATL, there is no option for using classes.

KerMeta is divided into two parts: a structure which corresponds to EMOF and a behavior which corresponds to actions.

Figure 5: EMOF extension and Kermeta promotion [Kermeta]

1.2.6. MOFScript
MOFScript is a model to text transformation tool, based on one of the MOF Model to Text Transformations to support code and document generation from models. MOFscript is metamodel independent; it accepts any kind of metamodel and its instances for text generation. MOFScript is now one part of the Eclipse Generative Model Transformation (GMT) suite of tools that is a collection of tools in Eclipse that supports MDE.

MOFScript plug-in has been developed in two different parts; Service component and tool component. Figure 6 shows the architecture of MOFScript. Service component provides functions for parsing and executing of transformation. Model Manager, located in Service component, is an EMF plug-in that manages the models in MOFScript.
Figure 6: MOFScript Component is divided into service component and tool component [MOFScript].

The Parser and Lexer is a service that parses the textual definition of transformation. Semantic checker is checking transformation correctness by validating the rules that are called. Execution Engine handles the executing of all transformation. Usually it accepts a model as input and generates some text file as output. The Traceability Manager manages the traceability between the original model and generated text and synchronizes changes made in the model and the related texts.

Figure 7: MOFScript plug-ins [MOFScript]
There are some other modules in tools component. The Editor gives some means to edit transformations and execute them. Result Manager handles the output of the transformation file and integrates them with Eclipse platform which mainly includes integrating generated Java code with Eclipse. The Outline Viewer, Problem Viewer and Preference Manager, give us some graphical interfaces for writing and executing transformations.

For deployment, MOFScript is available in five plug-ins for Eclipse.

1.2.7. MTF
IBM Model Transformation Framework (MTF) is an EMF based model transformation framework. It is now available at alpha works and provides a declarative means of specifying metamodel relationships, similar to that of QVT relations. The aim of MTF is to
simplify the transformation tools by supporting tractability and updating reconciliation. The announced most common use-case of MTF by IBM is as follows [IBM-MTF]:

- Model, code and document generation
- Model compare and merge
- Design pattern expansion
- Model consistency

Like MOFScript, MTF has been deployed in several Eclipse plug-ins. It gives transformation engine for EMF models, and an editor to define, debug and run rules. The engine depends only on EMF libraries and it is possible to be called in other JAVA based platforms as well as Eclipse.

In MTF model, transformation relies on EMF models. Rules are mappings between different objects of source and destination models. Declarative rules in MTF are also called “Relate” because of the relation that is made between objects.

Code below is a “relate” rule that is mapping an attribute.

```java
relate UmlAttribute2Column(
    uml:Property property
    when util:InstanceOf "uml:DataType" (property.type),
    rdb:Column column)
    when equals(property.name, column.name)
{
    equals(column.type, "VARCHAR")
}
```

Code 2) A source in MTD that shows mapping an attribute [IBM-MTF]

1.2.8. ATL

ATL Engine is another QVT like transformation language, developed by the INRIA Atlas team. ATL is one of the core technologies in the Eclipse M2M project. It is developed in a set of Eclipse plug-ins and works as a development IDE for transformations. ATL includes execution and debugging in the current version and is integrated with EMF and MDR. It is a transformation language that provides set of target models from source models.
ATL is a combination of declarative and imperative programming. Although it is aimed to define rules in declarative format there is the possibility to use imperative style as well. Figure 9 demonstrates basic concepts of ATL.

![ATL operational context](image)

**Figure 9: ATL operational context.** $M_t$ is a transformation file that conforms to ATL. $M_a$ is the input model for $M_t$ and $M_b$ is the output of the transformation. All metamodels conform to MOF.

ATL is deployed with the help of some wizards. One that is used for configuration of ATL files is shown in Figure 10. In that wizard a source model and a target model are specified respectively as Model a ($M_a$) and Model b ($M_b$) that conform to metamodel a ($MM_a$) and metamodel b ($MM_b$). The transformation that is created conforms to ATL. Source and destination models are defined by Ecore, KM3 or XMI. ATL is supported by several tools under Eclipse platform including debugger, compiler, virtual machine and editor.
In this chapter version management requirement for software development is discussed briefly. We describe Source Code Management (SCM) as a general purpose solution. Model Code Management (MCM) is presented as a solution for MDE.
To help demonstration of the concepts and solutions, a simple model of networks called NET model, is considered for the rest of this report. A NET model that is a simplified model for network devices and servers, may consist of several nodes and edges. Each edge has one start and one end node. Nodes can be connected to several edges while each edge is only connected to two nodes. In order to keep the examples simple no other limitations or assumptions have been applied. In the rest of this report we present the conflict model of this NET model as a solution to detect conflicts in MCMs. In this chapter, model versioning support for the NET model is illustrated and model developing and difference model is shown. Moreover, it is demonstrated the need for MCM in MDE.

3.1. Model developing

Similarly to source code developing, in model developing models are distributed to developers and each developer extends some parts. For saving the result some repositories are also used. As expected, there is a possibility of changing the same part of work by more than one developer. So for each language there is a need for a set of tools to support comparison, merging and manipulation of sources. This is especially necessary in the development and implementation phases of project where different developers work on the same program concurrently.

Figure 11 shows “Model A” of NET that is changed by two developers. As you can see changes to each model can be described by means of “Add”, “Delete” or “Change” operations on a node or an edge (Model elements) of NET.
For saving sources from problems in concurrent versions, developers update their programs via some Source Code Management (SCM) tools like CVS or Subversion. These tools compare the server version with the new version.

SCMs are faced with many challenges in order to make right sources. Merging is one of them. There may be some conflicts between two versions that are about to be merged. To demonstrate the problem of merging, we refer to the previous example of NET model. As shown in Figure 11, two versions (Version 1 and Version 2) are extended from the initial Model A. Both new developed versions are correct in a sense that there is no language error in each of them (Each one is defined in the language of NET so none of the developers gets compiler errors or modeling language errors).

Suppose that Version 1 is the first which is replaced with Model A in the source repository. Still there is no problem but then Version 2 is requesting the repository to be saved. Here the SCM faces a problem confirming it. Model A shows a network with three nodes and one edge. In Version 1, the “Server” node has been removed and as there is no modeling error (compiler error or language error), it can be uploaded to the MCM server. On the
other hand, another developer has added a new edge to existing model A. Version 2 model shows the added edge. The problem with these two versions is that one developer has added a new edge to the model that depends on an existing node when the other developer has deleted that node. In this case a conflict occurs. On next chapters we will return to this problem and solve it through a conflict detecting solution.

Figure 12: Process of updating new models in to a model repository. This architecture is described with details by Takafumi Oda and others [version].

Figure 12 shows the process of saving new models to the model repository. Methods check-in and check-out are used for each time that a new version is going to be saved [version].

As discussed so far, each model conforms to its own metamodel, therefore a language of that model should be created. It is usually created by some method engineer or other MDE experts. Then, the metamodel is converted to some plug-ins for model development IDE like Eclipse GMF, and then the model elements are ready to be used for generating models. Software Engineers use these elements, create the initial models and add them to
the model repositories. Other software developers check-out this program and extend the existing program of models. New versions of the existing model are saved into the model repository with Check-in method.

After using the check-in method, if a conflict occurs, the output of such a tool is usually presented in a way that developers can see the changes and decide the right decisions for replacing their programs with the existing one.

3.2. Difference Model

In order to save storage and reduce the costs of saving all parts of new versions, some version control systems are used. Version control systems keep only the changes made for each version. Changes are stored in difference models. There is no matured tool for calculating the difference model for all models in MDE but there are some tools for some models where they show differences between old version and new version by highlighting different components or elements. Rational Rose [Rational] and Konesa [konesa] have a graphic view for model differences. Coloring technique is used in several tools to show changes made by new versions [Coloring]. Figure 13 presents two versions of a UML model and Figure 14 shows a tool that demonstrates the difference of these two versions by highlighting some part of them [ModelDiff].

![Figure 13: Two version of a UML model](ModelDiff).
Antonio and others [Conflicts] have proposed an MDE based solution for representing model difference. As shown in Figure 15, for a real system there is a model that conforms to a metamodel and that metamodel conforms to another model which is called the Meta metamodel, like Meta Object Facility metamodel (MOF metamodel), ECore, KM3, and so on. The difference model of two input models is a model itself, and that consequently should conform to a difference metamodel. The difference metamodel is generated automatically from the metamodel the input models conform to. In order to generate this difference metamodel a transformation file called “Metamodel to Metamodel Difference” is written. The philosophy of difference metamodel is based on a simple idea that all changes can be classified to “Added”, “Deleted” and “Changed” actions that could happen to existing model elements of an input model. Therefore, the difference metamodel includes three extensions for each model element in the metamodel of the input model as shown later in Figure 19.
In order to illustrate the difference model and difference metamodel, we refer to the NET model. For the two versions of Model A shown in Figure 11, we create two difference models.

Figure 15: Difference Model of two input model conforms to Difference metamodel which is created automatically from the metamodel of input model [Conflicts]

Figure 16: Difference model of Version1 and Model A. Changes made for V1 and Server Node is deleted
Figure 16 shows changes made for making V1 out of Model A. As it is shown, a node which is named “Server” has been deleted. The “DeletedNode” is a new element of “Deleted” type and it inherits from “Node” as well. So it shows a “Node” which is deleted.

Similarly Figure 17 shows the difference between Model A and V2 in Figure 11 as a difference model. There is “ChangedNode” that is a “Changed” type and “AddedEdge” that inherits from added. Model element “ChangedNode” also has a reference to the new updated node which is Server Node. This node includes “Edge2” to its edges.

The model difference we have introduced can have two main usages in MDE field. The first usage is for version management tools that use difference of new codes and server codes and present them to developers in text editors. In case of MDE, basically models are compared and for model developers it is more applicable to show the differences again in model style rather than text style. So the presented model difference can be used in order to keep differences in server and to show them graphically to model developers. The second usage of this difference model is in conflict detection scenario. It helps to generate the conflict metamodel and finally helps us to detect conflicts between two versions. More details are discussed in later chapters.

### 3.3. Model Code Management
Source Code Management tools or SCMs are general purpose means for code management. They are applicable and used for documents, source codes, programs and other information and also for Models. Therefore, these tools are not efficient for being used as model code management. For instance, users (Model developers) should decide for simple problems, conflicts of new versions and existing version. As in MDE models are the heart of the system, it would be profitable if troubles and conflicts of models versioning are shown to the developers in form of models, as opposed to text formats. If we use traditional SCM, first there is a need to convert models into texts, then to use SCM and again convert results to model forms to be shown to developers. Currently some tools that use such techniques are available for some model domains like Argo UML [Argo]. Argo transforms a diagram into a XML document [Version], and after the transformation, developers can use text based version control tools such as RCS and CVS for transformed XML documents. Some other tools such as Eclipse UML, Jude and Poseidon have interfaces that enable them to use CVS or other SCMs [EclipseUML].

This solution in MDE does not follow the idea of “Everything is model”. In addition, the limited result that is received by text based comparison is not a good achievement. In a scene that integration with models is not stressed, there is a need for Model Code Management (MCM). MCM is then a platform which should be created for models management purpose. It can use existing code management tools but needs to handle models in general. Such a platform includes difference models as the basic elements for version management. It introduces conflicts in concurrent versions with the help of models. In this thesis we propose a structure for detecting conflicts that can also be integrated in MCMs later. Therefore benefits of MCMs are not only for model based versioning system but they also help developers with introducing conflicts in model format. The possibility to define conflicts in MCMs will help developers with the simplicity of problem detection. It is also possible to define an automated reconciliation strategy for conflict cases.
### 4. Difference Metamodel

In order to compare two models in MCM, we need to generate the difference of two input models. The concept of “Everything is model”, suggests us to present that difference in form of a model. The difference model should conform to a metamodel which we call Difference Metamodel (DMM). Some requirements for model difference are discussed in other researches and a metamodel is also suggested for covering them [Antonio]. That idea is based on three events that can happen for a model element. A model element can be deleted, changed or added in newer versions. If these three events are considered in new difference model, the DMM will be more likely an extension of the input metamodel with three extended elements for each existing element in input metamodel as discussed earlier.

To describe the difference metamodel, we use NET model which was introduced in previous chapter. In order to understand the concept, first metamodel of NET is studied and then difference metamodel of NET is shown.

Code below shows the NET metamodel defined in KM3 format:

```km3
package NET
{
class Node
{
    attribute name : String;
    reference edges : Edge;
}
class Edge
{
    attribute name : String;
    reference nodes : Node;
}
package PrimitiveTypes {
    datatype String;
}
```
Code 3: NET metamodel in KM3 format

NET metamodel includes two Node and Edge classes. It has also a primitive type of String. Classes in KM3 can have a reference of each other. In addition; some other attributes may be considered for each class. As shown; class Edge has an attribute “name” which is a string and a reference “nodes” to Node class. Also for the class Node, there is “name” of string and a reference “edges” to edge.

Figure 18 shows NET metamodel diagram. Each NET model can have different Nodes and Edges and each new version of an existing model can differ on “added”, “deleted” or “changed” in NET elements; that are “Node” and “Edge” as discussed.

Figure 18: NET metamodel diagram

Difference metamodel of NET should be able to keep the changes occurring in NET model revisions. It is rendered by adding three children to the current elements of NET metamodel with “Added”, “Deleted” and “Changed” types. Figure 19 shows NET difference metamodel.
Figure 19: Difference Metamodel of NET metamodel; each element of NET metamodel is extended to three other elements that inherits Deleted, Added, and Changed elements.

The following code is difference metamodel in KM3 format:

```java
package NETDiff {

    abstract class Added {
    }

    abstract class Deleted {
    }

    abstract class Changed {
    }

    class Node {
        attribute name : String;
        reference edges : Edge;
    }

    class Edge {
        attribute name : String;
        reference nodes : Node;
    }

    class AddedNode extends Node, Added {
```

~34~
Keyword “extends” is used in KM3 for inheritance. It is also possible to have multiple inheritances as shown in Code above. For instance “Deleted Edge” class, extends both “Edge” and “Deleted” classes.

5. Conflict Metamodel

Our solution to detect conflict is not dependent on a specific conflict language. Antonio and others [Conflicts] suggested a metamodel for conflict management that is based on the difference metamodel illustrated so far. It is a general conflict language, which we pick in order to generate the specific conflict language defining conflict patterns between any model revisions. There is a possibility to change or replace this conflict language with some other conflict metamodels. The current one contains left and right pattern definitions for two input difference models. Each pattern includes one or more difference elements. Patterns are described in OCL language. Difference elements are extracted from the
difference metamodel discussed earlier in chapter 4. More details about the conflict metamodel can be found in [Conflicts].

In order to add patterns of conflict to an existing model, we first generate the conflict metamodel of that model. This is done through an ATL transformation that receives two inputs. First input is the language of the difference models we want to generate the conflict metamodel for, while the second input is a generic conflict pattern language. The output of that transformation is a conflict metamodel tailored to the former input metamodel. Figure 20 shows generated conflict metamodel of NET model.

![Figure 20: NET Conflict Metamodel; the idea is based on defining left and right patterns for elements of two input versions of the same model that results in conflicts.](image)

In the conflict language, the left and right patterns are connected by a conflict relation. In order to describe complicated conflicts, it may be needed to extend or modify the current conflict language. OCL expression is considered to define models. For solving the
detected conflict a reconciliation strategy is added to the conflict language. This reconciliation strategy may contain some automatic responses to the conflicts.

In order to be able to detect conflicts shown in Figure 11 for NET “Model A” concurrent versions, we should define the right and left patterns of the two input models in the language of NET Conflict metamodel shown in Figure 20. Several solutions are possible for defining such a conflict. One solution is described in Figure 21. Note that no reconciliation strategy has been taken for this conflict because we want to keep this sample very simple and understandable.

**Figure 21: Conflict Patterns in the language of metamodel NET Conflict.** There may be many debates on conflict language and reconciliation strategy. Current conflict detecting solution is independent from conflict languages and also from metamodel of input model.

### 6. Conflict Detection Architecture

Based on difference and conflict metamodels that already have been presented, we can generate conflict metamodels for any input model which conforms to KM3. The conflict metamodel enables us to develop an Eclipse plug-in for defining conflict patterns that may happen in the modeling language we develop. All generated conflict patterns, defined
preferably by an expert, are then used in a conflict detector tool to detect conflicts of parallel developed versions of the same model. Figure 22 demonstrates a scenario for detecting conflicts for two input models that are obtained as revisions of the same model.

Figure 22: Conflict Detection scenario: for different versions of input Model A, the conflict patterns are used to detect conflicts. These conflict patterns are generated by an expert with some elements of conflict metamodel of the Model A. In order to have conflict metamodel of Model A, input metamodel of Model A and a conflict language is merged.

The model contains several actions and packages as follows:
Actions are:

- **Merge**
  This part receives a conflict metamodel definition and a metamodel of Model A and merges them into another model which is “Conflict metamodel of Model A”. This merge is done with the help of an ATL file.

- **Generate Eclipse Plug-in**
  This is done with Java application. It receives “conflict metamodel of Model A” as an input and generates an Eclipse plug-in for defining conflicts. This plug-in contains conflict language elements that enable developers to drag and drop them and define a set of conflicts that may happen in developing “Model A”.

- **Define Conflicts**
  Developers can define new conflicts with the help of the Eclipse plug-in generated from conflict metamodel. It is possible to define several patterns that result in conflicts. OCL language is used to define difference model elements of two input versions.

- **Detect Conflicts**
  It is done automatically with a tool that compares the patterns of left and right models with set of conflict patterns. If both sides are matched, then a conflict will be detected.

Packages are:

- **Conflict Metamodel definition**
  This can be a conflict metamodel earlier discussed in chapter 4 or any other conflict metamodel that is based on difference metamodel.

- **Input metamodel**
  Any metamodel that conforms to KM3 can be considered as an input metamodel. This is a metamodel of the model that we want to define conflicts for it.

- **Conflict Metamodel of the input**
  The result of Merge process is a Conflict metamodel of input metamodel.
• Eclipse Plug-in  
With the process of “Generate Eclipse Plug-in”, conflict metamodel is transferred into an Eclipse plug-in to enable developers for defining new conflict patterns.

• Conflict Patterns  
It contains patterns for each two input versions of Model A.

• Version 1 of Model A  
The right model input of Model A

• Version 2 of Model A  
The left model input of Model A

• Model A  
This is a model which we want to develop.

6.1. Advantages

The process here is to generate conflict metamodel and then to create conflict patterns. Developers can benefit from these conflict patterns specially if embedded in MCMs. When developers check in their models with server, the conflict patterns are compared to the models. If any match happens, then the conflict is shown to the developer in a form of model. The basic idea of “Everything is model" is used in all parts of this scenario including input models, difference models, conflict metamodel definitions, conflict pattern definitions and conflict result. Compared to general purpose SCMs which are text based tools, this methodology is based on models. It provides them conflict resolution that can be supported by an automatic reconciliation strategy. Developers can define custom conflict patterns in a case that they are not satisfied with the built in conflict patterns that are defined by experts. The language of conflict is also easy to modify. This adds flexibility to MCMs and benefits developers with less cost and time that may be needed for updating models in model repositories.
7. Implementation

The merge part has been done by ATLAS Transformation Language (ATL) languages. In the field of MDE, ATL does the transformation between models. ATL is a Model to Model (M2M) component, inside the Eclipse Modeling Project (EMP).

Figure 23 shows the project folders and a view of the Eclipse project that contains our source codes. It contains “Metamodel”, “Transformation”, “Model” and some other folders. Each of these folders contains specific files for this project for example “Transformation” folder contains all ATL files needed for transferring models to other models. “Metamodel To Metamodel Difference” and “Metamodel difference To Metamodel conflict” are also included in this folder.

![Figure 23: Eclipse project](image)
One part of implementation is the “Metamodel Difference To Metamodel conflict” transformation. Code 5 and Code 6 show two rules of this transformation.

```
Module MetamodelDifference2MetamodelConflict; -- Module Template
create OUT : KM3 from anyMetamodel:KM3 ;

rule Metamodel2Metamodel {
  from
    s: KM3!Metamodel
  to
    t: KM3!Metamodel(
      contents <- s.contents
    )
}
```

**Code 5** Modules for transformation models in ATL languages

ATL is not a pure imperative language, whereas it is a hybrid language embedding imperative rules inside declarative correspondences between source and target metamodels. The elements of input models are compared to the types of each rule. In a case that a match occurs, that rule will be executed. Only one rule should be matched for each set of entries.

Above code contains the main module (“MetamodelDifference2MetamodelConflict” module) developed for this thesis. The aim here is to add a conflict definition that is actually independent to the input difference metamodel (Difference metamodel model of any model is acceptable). The difference metamodel of a metamodel has been created automatically in the same solution from input metamodel. All metamodels conform to KM3 format. KM3 is available in the Eclipse platform. Ecore is another format that metamodels are usually defined with and in that case one transformation from Ecore to KM3 is needed in order to prepare them as inputs of transformation files.

Input of the main module could be a difference metamodel of any model and the output is the conflict metamodel of that model. First solution for generating conflict metamodel is to hardcoded the definition of the conflict language in the ATL file in a way that this definition will be added to the input difference metamodel and generates the related conflict
metamodel. This solution then is not independent from conflict language because any changes of the conflict language needs some changes in the ATL file. The other solution is to get two models as the input for the ATL file. One input model for the difference metamodel and another for the conflict definition. With the recent solution, the conflict languages can be changed any time and there is no need to change the transformation file.

The first solution is successfully implemented but the second one is done with an error of referring to the right model and still needs more efforts to be finished.

```rule
Rule Package2PackageDiff
{
    from
        s: KM3!Package
        {
            not (s.name='PrimitiveTypes')
        }
    to
        t: KM3!Package(
            name <- s.name+'_AfterTransfer',
            contents <- Sequence
            {
                ConflictBlock, PatternBox, BooleanRestriction,
                DifferenceElement, ConflictRelation, ConflictEndLink, MetaData, IntegerProperty,
                StringProperty, BooleanProperty,
                ReconciliationStrategy, Rule,
                OCLExpression, OCLCriteriaExpression, OCLRULEExpression,
                s.contents
            },
            metamodel<- s.metamodel
        )
}
```

**Code 6) Rules are activated if input models are matched with “from” part**

The code above shows the rule where new elements have been added to the target model. All new elements like “Conflict Block” are added to the main module.

Below the element “composedRules” is defined:
composedRules: KM3!Reference

(name<-'composedRules',
package<-s.package,
isContainer<-false,
owner<-OCLCriteriaExpression,
lower <- 0,
upper<- 1,
isOrdered <-false,
isUnique <-false,
type <-Rule
)

Code 7: Declaration of composedRules model in ATL; composedRules is Reference type in KM3

Ant files are other type of files that are used in this project. For running and building an ATL solution there is a need for a configuration that is usually saved in the Eclipse executable folders and is not attached to source files, therefore it will be lost in new Eclipse engine. To avoid setting the configuration file repeatedly and for better visualization of configuration setting, “Ant” files are used. Code 8 shows the ant file used in the project to run the “Difference Metamodel to conflict Metamodel” transformation. Source models and target models are defined in this file. Also there are some commands that can print some messages for “Console” window of the Eclipse during execution time which has been used.

<project name="BuildMetamodel" default="transformAll">
  <target name="transformAll" depends="loadModels">
    <antcall target="MetamodelDifference2MetamodelConflict" inheritall="true" inheritrefs="true">
      <param name="source" value="/ModelDifference/Metamodels/NETDiff.km3"/>
      <param name="target" value="/ModelDifference/Metamodels/NET_Conflict.km3"/>
      <param name="atl" value="/ModelDifference/Transformations/test.atl"/>
    </antcall>
  </target>

  <!--parameters: (sample : String) →
<target name="MetamodelDifference2MetamodelConflict">
  <echo message="Transforming ${source}"/>

  <!—Load source models →
  <am3.loadModel modelHandler="EMF" name="SimpleUMLModel" metamodel="KM3" path="${source}"/>

  <injector name="ebnf">
    <param name="name" value="KM3"/>
  </injector>
</am3.loadModel>

  <!—Transform KM3 model into KM3Conflict model →
  <am3.atl path="${atl}"/>

  <inModel name="simpleUML" model="SimpleUMLModel"/>
  <inModel name="KM3" model="KM3"/>

  <outModel name="OUT" model="KM3ConflictModel" metamodel="KM3"/>
</am3.atl>

  <!—Save target model →
  <am3.saveModel model="KM3ConflictModel" path="${target}"/>

  <extractor name="ebnf">
    <param name="format" value="KM3.tcs"/>
  </extractor>
</am3.saveModel>
</target>

<target name="loadModels">
  <!—Load ATL metamodel →
  <am3.loadModel modelHandler="EMF" name="ATL" metamodel="MOF" path="/ModelDifference/AMMATools/ATLT.ecore"/>

  <!—Load KM3 metamodel →
  <am3.loadModel modelHandler="EMF" name="KM3" metamodel="MOF" nsUri="http://www.eclipse.org/gmt/2005/KM3"/>

  <!—Load TCS metamodel →
  <am3.loadModel modelHandler="EMF" name="TCS" metamodel="MOF" path="/ModelDifference/AMMATools/TCS.ecore"/>

  <!—Load KM3-TCS model→
  <am3.loadModel modelHandler="EMF" name="KM3.tcs" metamodel="TCS" path="/ModelDifference/AMMATools/KM3.tcs">
8. Conclusion and future works

Inspired by the concept of “Everything is model”, we have proposed a conflict detection scenario that can be applied in MCMs. Some parts of Conflict Detection Architecture have been implemented while some other parts are left for future works.

Figure 24 shows the status of the project. Apart from the partial implemented parts and not implemented parts of the “Conflict Detecting Architecture”, which should be completed in future works, there are other possible improvements as follows:

- Optimizing the Conflict Languages.
  For the “Deleted Referenced” conflict that was discussed in previous chapter, as well as for other possible changes in the conflict patterns that result in the same conflicts, the conflict language should enable developers to define a general template for those simple conflicts that may happen for all models. The mentioned conflict is highly dependent on a specific model while these conflicts are common for all models.

- Developing special language for specifying the patterns and the reconciliation strategy.
  There are some limitations with the usage of OCL as mentioned in [Graphic]. A language that could be expressed in form of models is needed.

- Developing a framework for MCM.
  There should be a framework for MCM in a way different plug-ins like the conflict plug-in we proposed can be integrated with them.
Figure 24: Conflict detection scenario; the green parts are implemented in this thesis, the blue parts have been partially implemented and the red part should be considered in future works.

- Exploring the idea of automatically reconciliation strategy.

After conflict detection, an automatic reconciliation strategy is suggested that needs more research to be practical and should be studied in future works.
9. References


[MOFScript] Jon Oldevik; “MOFScript Eclipse Plug-In: Metamodel-Based Code Generation”; April 2006; available at 10.10.2009 at:


[ATL] Frédéric Jouault Freddy Allilaire and others; ATLAS group of Université de Nantes; “ATL: a QVT-like Transformation Language”; June 1007; ACM 1-59593-491-X/06/0010.