Universidad
Politécnica
de Cartagena

Hochschule Ulm
University of
Applied Sciences

University of Applied Sciences Ulm

Department of Computer Science

Master Course Information Systems

Collaborative Work on Distributed Models

Master Thesis

by

Ángel López Moya

Advisor:	Prof. Dr. Christian Schlegel
Project-Advisor:	Alex Lotz

Declaration of Originality

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Ulm, den 20. Jun 2013 Ángel López Moya

Abstract

The collaborative work on distributed models is a big challenge in the robotic industry. To fragment the system into smaller components ease the software development because every agent can be specialized in a specific part, but the collaborative work of the different agents in the complete system present challenges such as to ensure consistency. A technology that provides support to establish remote repositories and the functionalities to work with them ensuring consistency and security is needed. The first goal of this thesis is to analyze the main challenges that occur in this kind of system. The next goal then is to choose which of the main tools that allow to work on remote repositories can cover these necessities and to show what approach can be followed to cover the different features with the chosen technology. In the end, the previous analysis is discussed to show which level of completion is reached.

Acknowledgements

I want to start by thanking everyone who directly or indirectly contributed to this work in some way or another.

First of all, I want to thank to my project-advisor Alex Lotz who supported me along this thesis. He gives me lot of advices about how to focus on the work and he was very patient helping me with the document, giving me feedback and showing me other ways to improve the work.

I also want to thank everyone who work in ZAFH Servicerobotic Laboratory, specially to Matthias Lutz and Dennis Stampfer who accompanied my stay in Germany. Although I did not work with them directly, it was very interesting for me to see all the work that they made there.

I can not forget Christian Schlegel. He gave me the opportunity to make my thesis in his laboratory. In addition, he always kept me informed about the overall work that they were working on. It was interesting to learn the difficulties and challenges that a researching group has to face everyday.

Special thanks goes to Juanfran Ingles Romero who gives me his support from Cartagena and to Cristina Vicente-Chicote, without her I would not have chosen to come to Ulm.

Finally, my thanks to my family who always supported me during my studies and made it possible to I finish my master thesis in Ulm.

Contents

Chapter 1

Introduction

Software complexity, particularly in robotics, is still the main barrier to introduce a successful robot market. Many companies develop robotic systems from scratch. Although, many algorithms are available, it is challenging to integrate them into a consistent system. One approach to cope with this problem is to use Component Based Software Engineering (CBSE). Thereby a system is fragmented into software parts of manageable complexity, according to their individual concerns. To be able to collaboratively work on components in different companies, which are specialized on particular domains (like navigation, human-robot-interaction, etc.), it is necessary to explicate relevant component properties in a consistent way. Model Driving Engineering (MDE) provides the means and tools for this purpose. However, it is still challenging to distribute the models, while ensuring consistency.

1.1 Motivation

Nowadays models are presented in the entire lifecycle of software engineering projects. Not only as an abstraction to software design but also for code generatation. With this popular use of model-driven development in industry, the collaboration of multiples companies is an important necessity. The main goal of this thesis is to enable collaborative work on distributed model repositories with special focus on the robotic domain. Thus it is necessary to evaluate which requirements are needed (e.g. need for merging, conflict detection, versioning, etc.) and analyze different tools or projects which allow to work on distributed model repositories and then to choose the fittest one.

1.2 Examples of Use

This section shows some generic use-cases to demonstrate common situations with the purpose to show the challenges that have to be solved.

To explain this, it is better to imagine a big software which is being developed by several companies. Every company is in charge of a different parts of this software. All the companies use a standard way to define the structures, interfaces, etc. so the different pieces can fit together.

Each of these companies has to modify parts of the software and also needs access to other parts. For example, one company is a database expert and is responsible to create the database. Another company is the application domain expert and needs to modify some parts of the database to adapt it to the domain needs.

It is evident that there are different roles involved in the creation of the overall system. It has to be also some administrator who manage the infrastructure of the system. For example there are users who just need read access to some (or all) parts. Another users will need access right to write or modify specific parts of the model and all of this has to be managed.

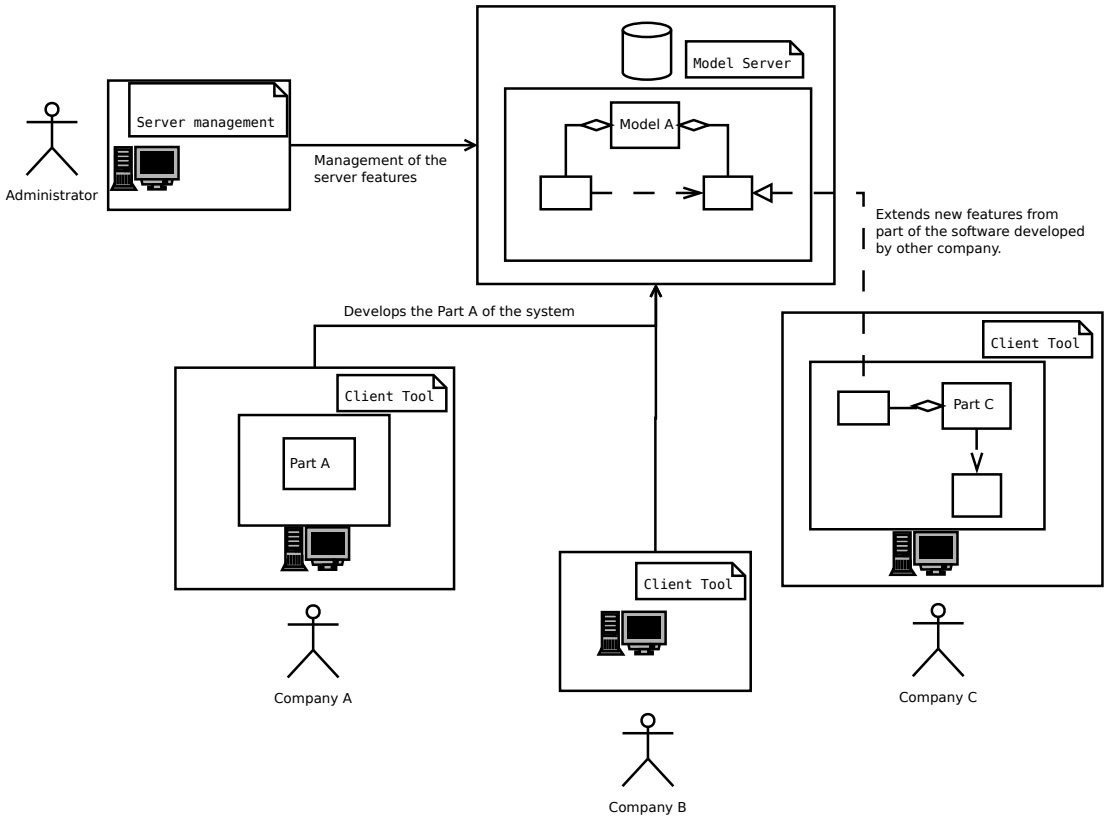

Figure 1.1: Basic Structure of the System

Figure 1.1 shows the general idea for the previous example. The example shows one model server and different companies connected to it. Each company works in different parts and they make different actions in the system. In addition there is an administrator who will manage the features in the server.

1.3 Challenges

In order to cover the examples presented in the previous section, and having in mind the figure 1.1 the following challenges need to be addressed.

- From the example, when a company wants to modify some parts of the software, it should be possible to restrict the access to only registered users from this company. Another feature is to have some kind of user interface to be able to work with the software. One of the challenges is to manage the way how different companies can save changes in the server as well as to control what kind of users can access to these models. In a company for example there are different departments and it is needed to have rights or permissions for access, modification, etc.

  In this example, it is also evident that to achieve consistency some mechanism is required to avoid that various users could modify parts of the software at the same time and create a conflict (e.g. in figure 1.1 the Company A and B can modify the same part and it will provoke inconsistency). In addition, it would be necessary to have some kind of notification system, to advice the other users or companies that one specific part of the software is being modified at the moment.

- In the example, an interesting feature is to make references to parts of the software which is stored in a remote repository. The system has to support these remote references and to notify the changes in these parts to the users (e.g. in figure 1.1 Company C is referencing some part of the software in the server to extend new capabilities).

- As it has been commented in the example, it will be necessary to have an administrator who will be able to change passwords, access rights and other necessary actions for the server side. It is necessary to have an interface to manage all this features in an easy way.

1.4 Organization and Approach of this Thesis

The next chapters of this thesis are organized as follows:

1. **Chapter 2**, there will be an introduction to Model-Driven Engineering, to explain the basic concepts of this software development methodology. In addition the chapter will analyze two different Eclipse Tools which allow collaborative work on distributed model repositories. The last part will show a comparison between this two tools and to choose the most fitting for our case.

2. **Chapter 3** will analyze the specific case of communication objects.

3. **Chapter 4** will analyze specific the features and requirements to be solved. After that, with the chosen tool, the chapter will show how this tool can cover the requirements and to provide solutions for the missing features.

4. **Chapter 5** will summarize to what extent the objective of this thesis has been solved.

5. **Chapter 6** will show some conclusions and what are the next steps to improve the collaborative work on distributed model repositories.

Chapter 2

Related Work

The focus of this thesis is on the collaborative work on distributed model repositories. This chapter will introduce the basic knowledge about Model-Driven Engineering, an analysis of two Eclipse Tools which allow collaborative work on distributed model repositories and a comparative analysis between these two tools.

2.1 Model-Driven Engineering (MDE)

Model-Driven Engineering [Sch06] (MDE) is a software development methodology which is focused on increasing the level of abstraction and automation in program development. The approach is to simplify the design process, encouraging the collaboration between different teams or departments. Thereby the most effort goes to develop a concrete domain model that afterwards will produce working software. That will be possible with model to model transformations (M2M) and model to text transformations (M2T).

2.1.1 Basic Concepts

The model-driven software development(MDSD) [Gro09] comprise a set of tools and technologies which allow first to develop a specific model language to help on the design of whatever kind of systems and afterwards to use a set of automatic transformations to obtain a final code for the applications. MDSD works around the definition, the use of models and transformations along the entire life-cycle of software development.

A model is a representation of reality. A model represents structures, behaviours and features. A system can be built with one or more models which will represent different aspects of the system in different levels of abstraction. So it is possible to create analysis models, design models and models very close to the development platform. The high level models can evolve to low level models with M2M transformations until the model is enough detailed to be able to generate code with a M2T transformation. Thereby, the model or set of models which represent the system together with M2M and M2T transformations allow to automate the process of software development. It is noteworthy that the generation of the entire system is in many cases not

feasible and even not necessary. Instead it is more important to create models with the right abstraction level such that these models can be easily understood by corresponding designers.

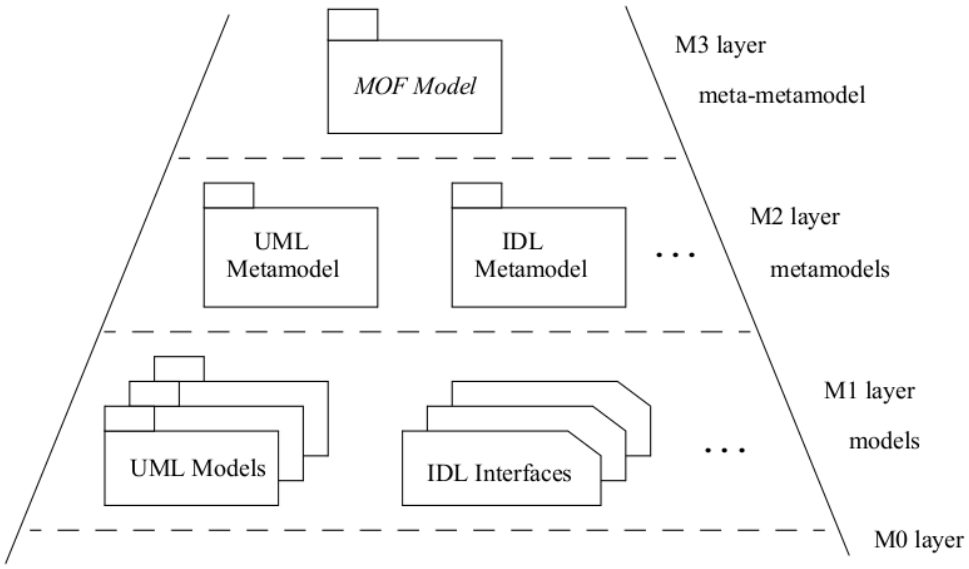

Figure 2.1: MOF standard pyramid[1]

Model languages define the syntax for models. Meta-models define the abstract syntax of model languages. They gather the concepts (words of the language) and rules that tell how to combine different concepts in order to create valid models. Figure 2.1 shows how a meta-meta-model allows to define a language that at the same time allows to define other languages. To avoid accumulation of too much "meta", the same metalanguage defines itself on this level in a reflexive way. The pyramid of the figure 2.1 belongs to standard MOF (MetaObject Facility Specification)[mof05] of OMG (Object Management Group)[omg03].

Nowadays MDE is a paradigm of software development with a lot of popularity in the software engineering field. The recent popularity of this perspective in the last years has been pushed by OMG and their initiative MDA (Model-Driven Architecture)[omg03] and the release of tools which support this approach, this allows to exploit all the potential of MDE.

Some of the tools that nowadays support MDE are: DSL Tools (Microsoft)[2], Meta-Edit+[3] (Meta-Case) and Eclipse[4]. The latter is an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the life-cycle. Furthermore Eclipse is a not-profit organization tool. In last years, Eclipse has become a de facto standard because it supports the main technologies from OMG.

[1]MOF: Meta Object Facility (MOF) Specification Version 1.4.1

[2]DSL Tools(Microsoft), currently Visual Studio Visualization and Modeling SDK: `http://archive.msdn.microsoft.com/vsvmsdk`

[3]Meta-Edit+: `http://www.metacase.com/products.html`

[4]Eclipse: `http://www.eclipse.org/`

2.1.2 Eclipse Modeling Project(EMP)

Eclipse Modeling Project is mainly a group of tools related to modeling and Model Driven Software Development (MDSD) [Gro09]. The purpose of creating this collection of tools is to coordinate MDSD technologies within Eclipse. The EMP is organized in projects which have to face the following capabilities: abstract syntax development, concrete syntax development, model-to-model transformation, and model-to-text transformation.

Figure 2.2 shows the structure of the EMP and also the features of the platform with Eclipse Modeling Framework (EMF) [FB03] as the core. The main purpose of EMF is to support abstract syntax development. The next layer, EMF Query, Validation and Transformation provides the management of model instances. Around these components are the model transformation technologies (M2M and M2T). In the last layer there are tools for concrete syntax development, GMF (Graphic Modeling Framework)[5] for graphic model representation and TMF(Textual Modeling Framework)[6] for textual model representation. Finally there are some elements orbiting around the core. These elements are projects which are focused on extending the capabilities of the platform.

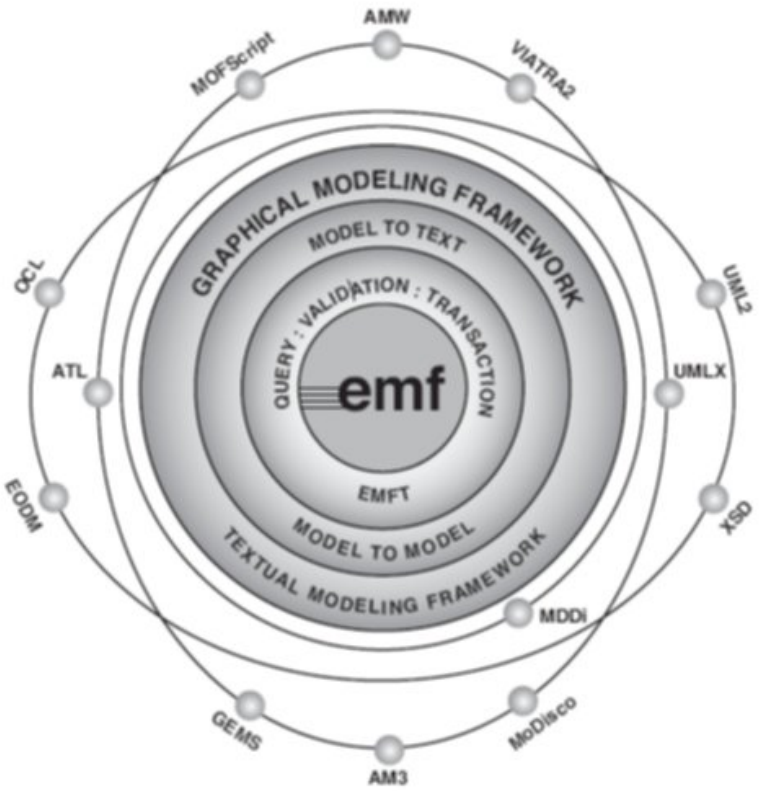

Figure 2.2: Eclipse Modeling Project[Gro09]

[5]Graphic Modeling Framework: `http://www.eclipse.org/modeling/gmp/`
[6]Textual Modeling Framework: `http://www.eclipse.org/modeling/tmf/`

2.1.3 Abstract Syntax of a Language with EMF

This subsection describes the basics on developing the abstract syntax of a DSL (Domain Specific Language) using the EMF framework (e.g. remember that EMF is the core of MDSD in Eclipse). This process covers from the creation of the meta-model to the generation of its supporting code in Java.

EMF is a modeling framework which allows to generate code to build tools and model based applications. EMF unifies Java, XML and UML[uml11]. Imagine for example that an application to manage a specific structure of XML messages is needed. The process would be to create a UML diagram from the initial schema, to develop a set of Java classes to manipulate the XML implementation and finally to generate an editor to use these messages. All this is possible with EMF. In addition, the EMF model can be defined using any of these tools independent of the others.

Next, the main features of EMF are described.

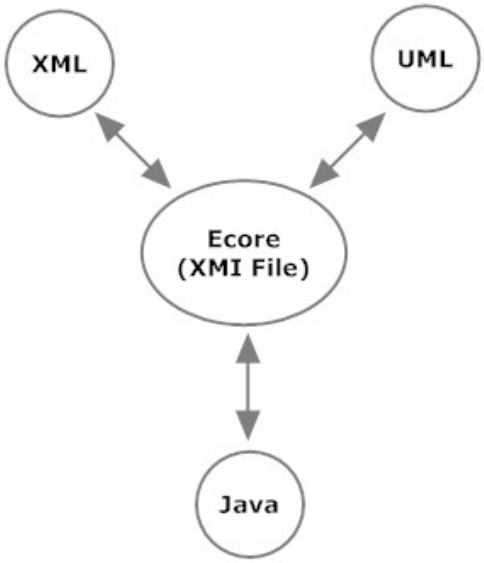

Figure 2.3: EMF unifies Java, XML and UML [FB03]

1. **Ecore meta-model**

   As a meta-model defines the abstract syntax of a language, providing the concepts of the language and the relationships between them, in EMF, meta-models are specified in terms of a simplified version of MOF (e.g. the meta-meta-model, see figure 2.1) called Ecore.

   Figure 2.3 shows a diagram that defines the main parts to describe an Ecore model. These parts are:

   - **EPackage:** Represents the package which contains the elements of the model (e.g. “box-arrow” in figure 2.4).

- **EClass:** Represents the modeled class. It has a name, zero or more attributes and zero or more interfaces (e.g. “Root”, “Box, “Arrow” in figure 2.4).
- **EAttribute:** Represents the relationship between modeled attributes. Each attribute has a name and a type (e.g. the attribute “name” of the class “Box” in figure 2.4).
- **EReference:** Represents the relationship between classes. A reference has a name, a boolean flag to indicate if it is a container relation and a reference (target) to other class (e.g. attribute “target” of class “Arrow” in figure 2.4).
- **EDataType:** Represents the data type. A data type can be a primitive type such as: integer, float, etc. or an object like java.util.Date type.

2. **Creating a meta-model**

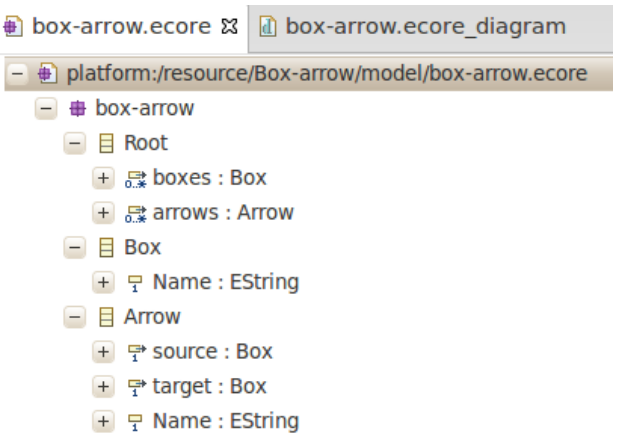

Figure 2.4: Abstract syntax in the tree-editor

EMF allows creating a meta-model in several ways. First, one can use the EMF tree editor to manage the different elements of the meta-model (e.g. to insert a new EClass) and the properties view to configure their features (e.g. to name the new EClass). Moreover, EMF also provides the Ecore diagram editor to define meta-models graphically. Apart from editors, it is possible to specify a meta-model by importing a UML2 model or by annotating Java classes, such that, a user class may identify an EClass of the meta-model. Normally, the meta-model is saved in a file with extension *.ecore. This file can be opened with a text editor to show its XMI (XML Metadata Interchange [**?**]) serialization (e.g., see Listing 2.1).

Following, we show an example of how to create a meta-model using the EMF tree editor. We use a primitive form of component model, thus, after generating the language infrastructure with EMF, users will be able to represent (in a tree editor) a set of named boxes interconnected with arrows. As shown in Figure 2.4, the meta-model has two main classes (EClass): "Box" and "Arrow". Each one with an attribute (EAttribute): "name".

In addition, Arrow has two references (EReference): "source" and "target", which allow connecting boxes. It is worth highlighting that we have included a base class, "Root", to contain the rest of the elements. As consequence of the serialization process in EMF, which is driven by containment relationships, there should be a (direct or indirect) containment between every EClass of the meta-model and the one rooting. Figure 2.5 shows the meta-model designed for the example in the Ecore diagram editor. Note that its appearance is similar to UML class diagrams.

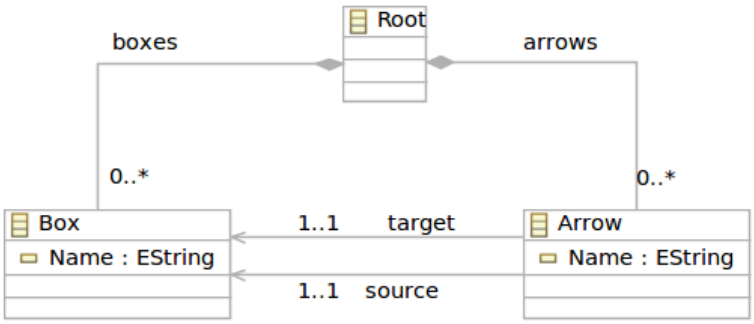

Figure 2.5: Abstract syntax in the ecore diagram editor

Listing 2.1 shows the Ecore model represented as XMI code.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="box-arrow">
5    <eClassifiers xsi:type="ecore:EClass" name="Root">
6      <eStructuralFeatures xsi:type="ecore:EReference" name="boxes" upperBound="-1"
7          eType="#//Box" containment="true"/>
8      <eStructuralFeatures xsi:type="ecore:EReference" name="arrows" upperBound="-1"
9          eType="#//Arrow" containment="true"/>
10   </eClassifiers>
11   <eClassifiers xsi:type="ecore:EClass" name="Box">
12     <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" lowerBound="1"
           eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
13   </eClassifiers>
14   <eClassifiers xsi:type="ecore:EClass" name="Arrow">
15     <eStructuralFeatures xsi:type="ecore:EReference" name="source" lowerBound="1"
16         eType="#//Box"/>
17     <eStructuralFeatures xsi:type="ecore:EReference" name="target" lowerBound="1"
18         eType="#//Box"/>
19     <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" lowerBound="1"
           eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
20   </eClassifiers>
21 </ecore:EPackage>
```

Listing 2.1: XMI code from .ecore

3. **Generating Code**

   Once we have defined the meta-model, EMF allows us to automatically generate an Eclipse plug-in that basically integrates the Java implementation of the meta-model and a tree editor for our language (for modeling boxes and their arrows). To get this Eclipse plug-in, it is necessary to create a *.genmodel* file. Then, from the *.genmodel* file, it is possible to generate the code for the elements of the model and the plug-in code. Figure 2.6 shows a model instance created with the generated tree-editor for the previous meta-model.

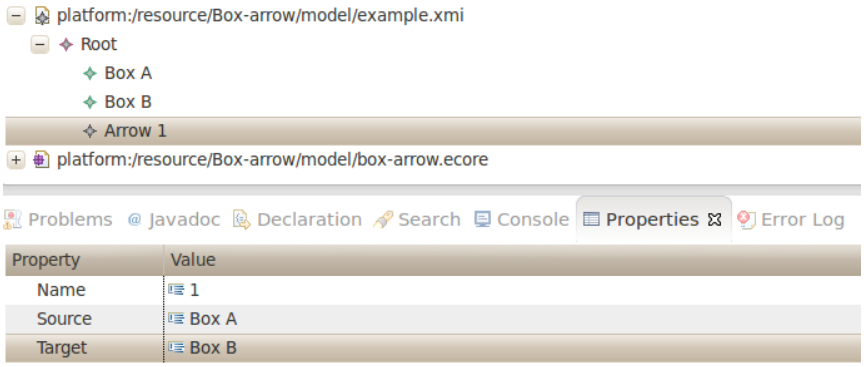

Figure 2.6: Abstract syntax in the ecore diagram editor

Listing 2.2 shows the model of the example in figure 2.6 by opening the model with a text editor.

```
1
2  <?xml version="1.0" encoding="ASCII"?>
3  <box-arrow:Root
4      xmi:version="2.0"
5      xmlns:xmi="http://www.omg.org/XMI"
6      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7      xmlns:box-arrow="box-arrow"
8      xsi:schemaLocation="box-arrow box-arrow.ecore">
9    <boxes Name="A"/>
10   <boxes Name="B"/>
11   <arrows source="//@boxes.0"
12       target="//@boxes.1"
13       Name="1"/>
14 </box-arrow:Root>
```

Listing 2.2: XMI code from model example

2.2 Eclipse Modeling Framework Store (EMF Store)

EMF Store[7] is a framework that enables the development of repositories to distribute, store and work collaboratively on models based on EMF. Different users can work offline, modify models and keep these copies locally in their workspace. When users want to send their changes to the

[7]EMF Store: `http://eclipse.org/emfstore/`

server, EMF Store will check that there are no conflicts between the different users in order to maintain consistency of the system. EMF Store provides a mechanism to allow users to solve conflicts interactively through an approach based on model merging for reconciling model versions. The server will keep a history of the different states of the models, so it is possible to change between different versions or to come back to a previous state. EMF Store provide features to merge, commit and update the models. It is possible to modify the behaviour of these features. EMF Store also provides a migration mechanism. In addition, EMF Store is easy to integrate in other Eclipse applications. In the following, the different features of EMF Store will be explained with more details.

The functionality of EMF Store can be summarized by the next points:

- **Automatic Persistence:** Every change performed in a model instance on the client side is directly persistent (without the need to have any contact with the server side).
- **Transparency on Model API:** EMF Store features can be integrated with EMF Client Platform[8]. In addition it is possible to create user interfaces for modifying and viewing models and by default EMFStore provides user interfaces for essential options such as history browser, synchronize models and update state of models.
- **Offline Mode:** EMF Store allows clients to work disconnected. Users can work offline and only require a connection with the server to update or commit changes.
- **Interactive Model Merging:** As previously mentioned, when several users notify changes on the same model simultaneously, EMF Store will detect the conflicts and start an interactive process for the users to solve the model inconsistencies. This process is performed by selecting the different merging alternatives.
- **Model Migration:** When a user changes a model, EMF Store can generate a migrator to update existing model instances to conform to the new version. In order to accomplish the migration, EMF Store relies on the technology of the Edapt Eclipse Project[9] (formally COPE).
- **Versioning:** Every time a user commits a change to the server, EMF Store will create a new version of the model. A user can recover an old version or just check the differences between versions. EMF Store provides a complete history of every change with user name, time and comments.

2.3 Connected Data Objects (CDO)

Connected Data Objects (CDO)[10] is a framework to develop EMF model repositories. It allows to work online in a collaborative way, and offers a run-time persistence framework.

[8]EMF Client Platform: `http://eclipse.org/emfclient/`
[9]Edapt Eclipse Project: `http://www.eclipse.org/edapt/`
[10]Connected Data Objects: `http://www.eclipse.org/cdo/`

CDO supports many deployments such as embedded repositories, offline clones or replicated clusters.

2.3.1 Functionality

The functionality of CDO can be summarized by the following points:

- Persistence: CDO abstracts the database technology, which allows storing models without being coupled with a specific database interface.

- CDO provides a mechanism for locking models in a repository. This prevent models from being corrupted or invalidated when multiple users try to write to the same model. Any user can modify those models to which they have applied a lock that gives them exclusive access until the lock is released.

- CDO offers a special view which enables users to see the history of changes in models. This view does not allow users to make modifications to the models. This feature can be enabled or disabled for each repository separately.

- CDO provides a very good scalability. It is achieved by loading just the objects on demand in the application. When objects are not being referenced by the application, they are removed by a garbage collector when the memory is low. In addition, there are some strategies about which fetch rules will be used.

- Collaborative work on models using CDO is transparent for the user, because the application is notified about remote changes in the model. Therefore, the users have the impression of working on the same instance of the object. The notification policies can be customized to obtain a specific behaviour, or even adding new handlers in the asynchronous CDO protocol.

- Fault tolerance is performed in different ways, using fail-over repositories controlled by a monitor or using offline branches and a session reconnection mechanism that allows user to keep working on models even when the connection fails or the server repository is down.

- Offline work with models is supported by two different mechanisms:

    1. By creating a local clone of a repository, including all history and branches. This repository is synchronized with the remote repository as soon as the connection is restored.
    2. By using branch points of the repository when the connection between the user and the repository is lost. After reconnection, the different branches will have to be merged.

- An interesting feature is to integrate the functionalities of CDO with other editor tools (e.g. GMF or Xtext). Currently, there is an Eclipse project called Dawn[11]. This project is part of the CDO project and provides an API that allows building extensions to use your current editors with CDO. At first it was developed just for GMF editors but nowadays Dawn supports other EMF editors such as tree-editor or Graphiti[12]. The user can keep working with his editors as he was working in a local file system. It is transparent for the user. On the other hand, Dawn has as objective the integration with web components to be able to work using a web browser and access to the models from any part of the world. In addition, Dawn has the future objective to integrate textual editors with CDO like XText.

2.3.2 Architecture

The architecture of CDO comprises client applications and the repositories. These two entities communicate with each other using the application level CDO protocol which can be used over different physical transport layers. Although CDO is designed for being used with the OSGi platform[13], it can work in stand-alone or even use other kind of containers, like application servers.

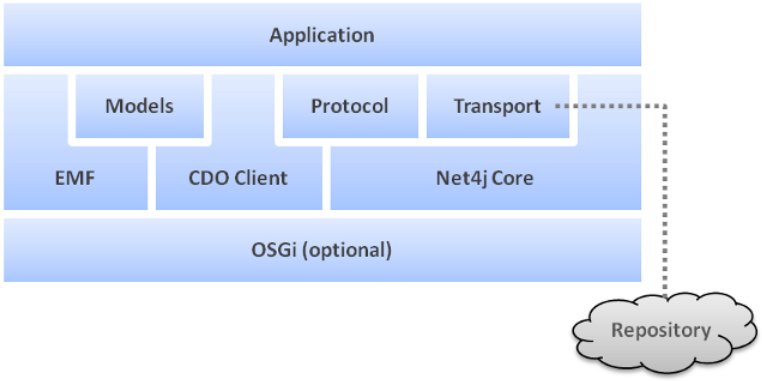

Figure 2.7: Architecture of a CDO application[14]

2.3.2.1 Client Architecture

The architecture of a CDO application has a strong dependency to EMF, because CDO model objects are EObjects. The basic functionality of CDO is integrated transparently with an EMF extension mechanism, but, in order to use advance functions, it is necessary to add dependencies into the CDO application.

[11]CDO Dawn: `http://wiki.eclipse.org/Dawn`
[12]Graphiti framework: `http://www.eclipse.org/graphiti/`
[13]OSGi: `http://www.eclipse.org/osgi/`
[14]CDO Architecture: `http://www.eclipse.org/cdo/documentation/` last visited: 4/26/2013

Figure 2.7 shows the architecture of a CDO application. As can be seen, the OSGi block is optional. Therefore the core components do not require OSGi, they can work stand-alone but the configuration is a bit simpler with OSGi. The Net4j[15] Core is a communication framework. It helps on the development of application protocols independent of the physical transport medium. Net4j supports TCP, SSL and HTTP transport protocols.

2.3.2.2 Repository Architecture

Figure 2.8 shows the architecture of the repository. There is a main block, the repository layer. The client application interacts with the repository using the transport block which has been commented in the previous subsection.

One of the interesting blocks of the architecture is the CDO Server Core. This is formed by the repositories where each one has various components such as revision manager, branch manager, package register, lock manager, session manager and commit info manager. Another interesting block is the CDO Store, which allows connecting to different kinds of databases such as JDBC databases, Hibernate[16], Objectivity/DB, MongoDB or DB4O.

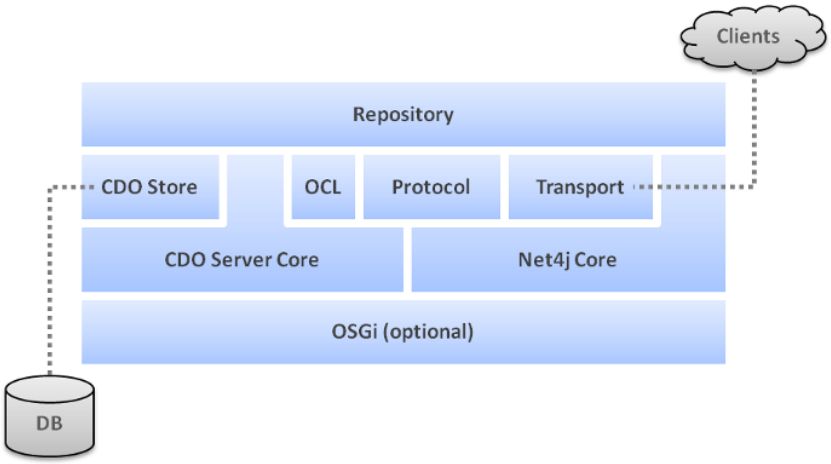

Figure 2.8: Architecture of a CDO repository[17]

2.4 Comparative between CDO and EMF Store

As CDO and EMF Store are the two technologies available in the context of distributed models, it is interesting to compare both and see which of them fits better for this thesis. The comparative is going to focus on different aspects or features that both technologies share or resolve in a different way.

[15]Net4j Signalling Platform: `http://wiki.eclipse.org/Net4j`
[16]Hibernate: `http://www.hibernate.org/`
[17]CDO Architecture: `http://www.eclipse.org/cdo/documentation/` last visited: 4/26/2013

- **Offline or online work**

  As the name suggests (Connected Data Objects), CDO is oriented on connected operation between the clients and the server. In contrast to CDO EMFStore is targeted at offline operation. EMF Store works much like SVN[18], one can checkout a model and commit and update changes of the model. The clients are not connected with the repository permanently, they just need connection to update the models or to commit changes. CDO offers a mechanism to work offline, which however was recently introduced and is not yet matured.

- **Branching and merging**

  In EMF Store, users are able to create a branch with the current version of the model and evolve it independently. EMF Store offers a mechanism to merge the different branches with a visual tool. On the other hand, CDO supports branching using a specific API which is used specially combined with the offline capabilities.

- **Migration**

  EMF Store has a complete mechanism for model migration. When a user changes something in a meta-model of the application, EMF Store provides support to migrate the instances of the meta-model. Several basic operations such as adding attributes or classes are handled automatically. For other functions, EMF Store helps to generate a migration code.

- **Scaling**

  CDO has been designed for high scalability. CDO uses a mechanism which only loads required parts of a model into the memory. A garbage collector additionally frees the memory for not used objects.

  On the other hand, EMF Store loads all models into the memory and does not have an own memory management. Thus, for large models it has high memory demands.

To sum up, both EMF Store and CDO provide the needed features and are thus valid alternatives for this thesis. However, CDO has a higher maturity compared with EMF Store, and is thus used from now in this thesis as the underlying technology. Both projects are starting to share technologies, for example, it is planned to implement a user interface for editing models to be used together with CDO and EMF Store, with the purpose to easily integrate applications. A lot of activities and discussions are going on in both communities. Some members claim that both projects will fuse into one shared project, but this is to be shown by the future.

[18] Subversion SVN: `http://subversion.apache.org/`

Chapter 3

The Robotics Case Study

The problem of collaborative work on distributed models is generic and independent of any particular domain. However, in order to better understand the real challenges and requirements for a solution (as presented in chapter 4), a real world example from the robotics domain is presented. This chapter is structured as follows.

First, there is an introduction about communication objects. Second, the section shows some use-cases related to communication objects which help in understanding which requirements will be necessary.

3.1 Introduction

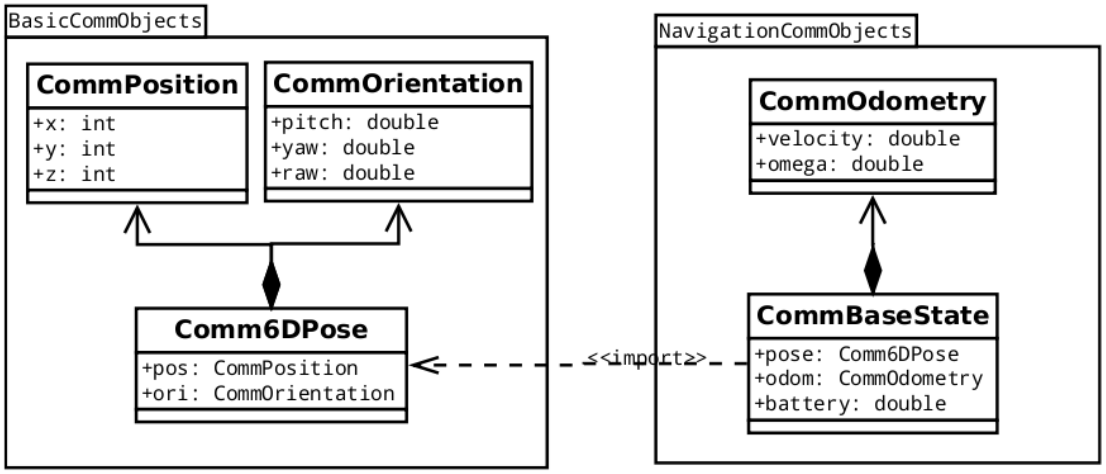

Figure 3.1: Communication Object Example

This thesis uses the robotic framework SMARTSOFT [SSL12] for the implementation. SMART-

SOFT defines the approach how to develop robotic software systems in a building blocks manner (using CBSE and MDE) [SSL12]. At the same time SMARTSOFT is an implementation of a robotics middleware[1] which abstracts over vendor specific communication mechanism and provides a set of communication patterns. However the approach in this thesis is independent of SMARTSOFT and can be used with other frameworks.

SMARTSOFT defines a component model which is implemented as the SmartMARS [SSL12] UML profile. The relevant parts of the profile are the component definition and the service definition. Services are used to exchange data between components. In order to define a service for a component it is necessary to choose the message type which is provided/requested by the service. In SMARTSOFT such message type are called communication objects (see figure 3.1). Technically, a communication object is just a class that defines the data structure for communication as a set of parameters. Each parameter can be either a primitive type (like int, double, string) or a reference to another communication object (which allows to create nested communication objects). Communication objects are stored in communication object repositories.

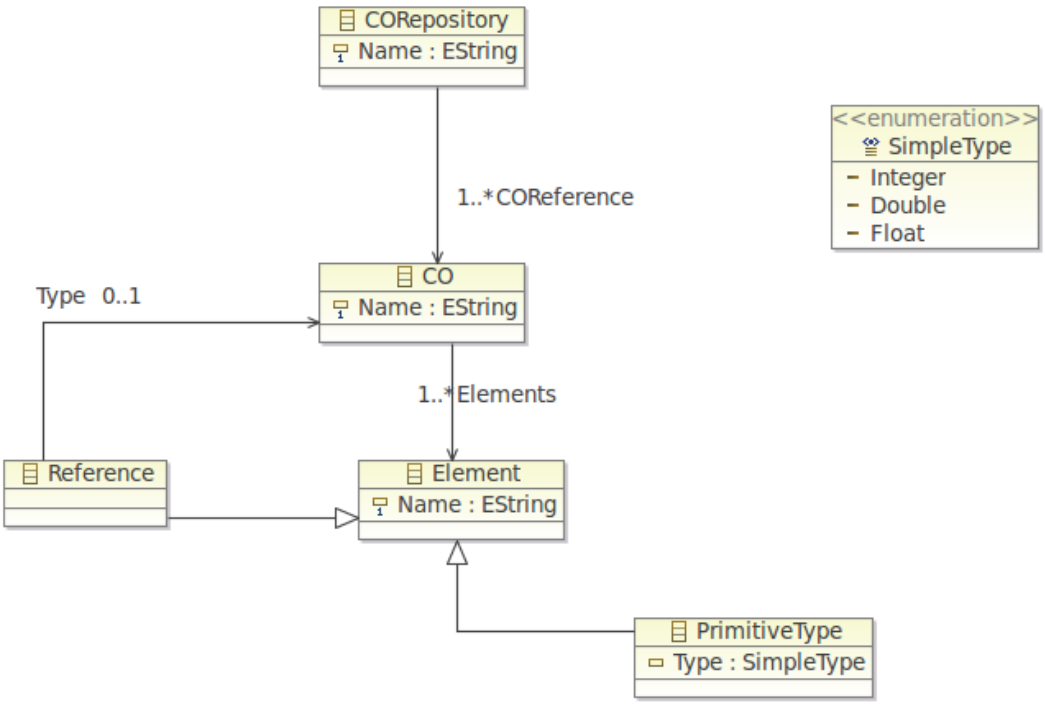

Figure 3.2: Communication Object Meta-model Example

[1] SMARTSOFT : http://smart-robotics.sourceforge.net

Figure 3.2 shows the Ecore meta-model of a communication objects repository. The root element is the repository (CORepository) which contains communication objects (CO). Communication objects consist of Elements. An Element can be either a primitive type (integer, double, string, etc.) or a reference to another Communication Object. The referenced communication object can be either in the same repository or in a different repository.

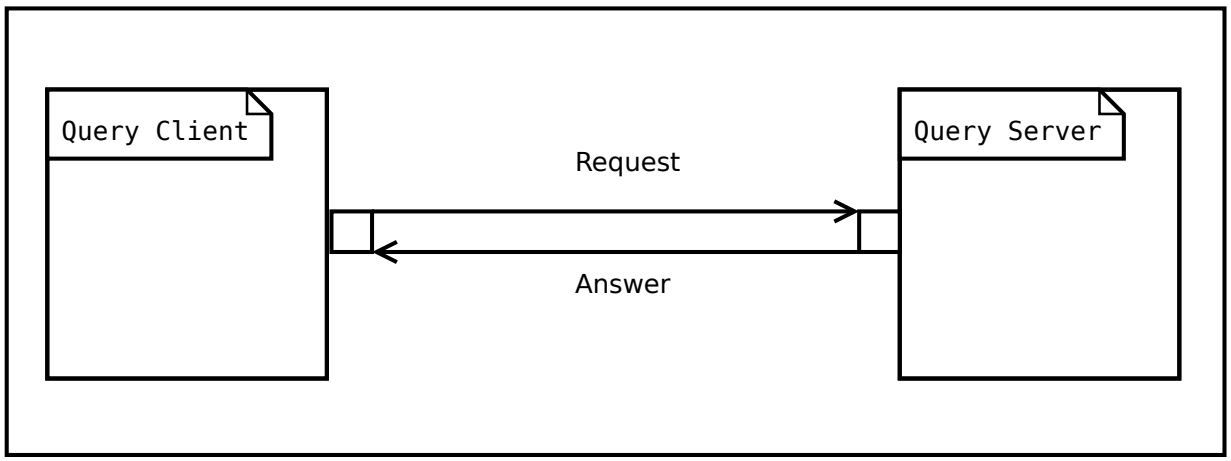

Figure 3.3: Communication Object Example

Figure 3.3 shows a simple example with two components. Between them, there is a communication based on the query communication pattern. A communication pattern will define the communication semantics. The semantics define the communication policy and how many communication objects are involved. In the case of communication pattern Query, the communication semantics is that each Query Client can send several Query Requests. For each Query Request the Query Server responds with a Query Answer. Thus the communication policy for a Query is request-response. There are two communication objects involved, the request and the answer. For these two communication objects any concrete communication object from a communication objects repository can be selected.

Now, with the focus on the communication objects, the different parts can be described. Figure 3.1 shows an example consisting of two communication object repositories, the “CommBasicObjects” and “CommNavigationObjects”. Both repositories contain simple communication objects like “CommPosition”, “CommOrientation” and “CommOdometry”. Simple communication objects just consist of primitive data types. Both repositories also have complex (e.g. nested) communication objects. Such objects nest other objects by referencing them. The referenced objects can be either in the same repository as with “CommBaseState” and “CommOdometry” or the referenced objects can be in a different repository as with “CommBaseState” and “Comm6DPose”.

3.2 Use-cases

Communication objects are part of service definition for software components. Thus the creation and definition of communication objects is typically a result of collaborative work or in other words, it is something that several involved parties (e.g. companies) must agree upon. An agreed set of communication objects can result in a standard which fosters reusability and exchangeability of software components in different scenarios and applications. In order to ease a coordinated creation of such communication objects, a tool support is required which helps to overcome the following situations:

1. Modification of existing communication objects in a remote repository.

   Preconditions:

   A communication object repository already exists.

   Tasks:

   Some of the communication objects in the repository must be modified (e.g. a new sensor type is invented that leads to new components with services which require to add new communication objects or to modify already existing communication objects).

   Constraints:

   - **Model Consistency:** A tool should prevent concurrent modification at the same time of one particular communication object by several users. This could otherwise lead to inconsistent models.
   - **Download/upload models:** During the whole process of modifying a model by a user, we must guarantee that this user is working with the latest version of the model. As a consequence, the process of downloading the communication object from the repository, modifying the model and finally uploading it, must be considered as an atomic operation.
   - **Notification and synchronization of models:** After a modification, all involved parties must be informed about the change and their local copies (if any) must be synchronized.
   - **Security access:** In order to ensure the identity of each user it is necessary to have a log-in system.

   In the following a practical example of how a user should proceed is shown.

   This first situation takes place when a user is going to modify a communication object in the remote repository.

   Steps:

   (a) User connects to a remote repository.

   (b) User has to log in with a name and password in the repository.

(c) User has to access to the repository where the communication object is stored.

(d) User modifies the communication object.

(e) User commits changes to the server.

(f) User closes the connection with the server.

The second situation occurs when a user is going to upload a model that has been previously created.

Steps:

(a) User creates a communication object model locally.

(b) User connects to a server.

(c) User has to log in with a name and password in the repository.

(d) User imports the model in the server using an URI from the workspace or the file system.

(e) User commits changes to the server.

(f) User closes the connection with the server.

2. Nested communication objects crossing repository boundaries.

In a typical market several companies exist which focus on particular domains. In a potential robotics market, some companies could be experts on mobile manipulation, other companies could be experts on mobile navigation or human-robot-interaction. It is clear that it makes sense to reuse generic communication objects (e.g. Position) in all these companies in order to create more complex communication objects (e.g. a Person object including a position of this person). Thus a mechanism is required to compose complex (nested) communication objects using other communication objects, which could be either local or even imported from other remote repositories (different to the current one). An example of this is shown in figure 3.1.

Preconditions:

Different communication object repositories already exist.

Tasks:

A new more complex communication object is needed. Thus it is necessary to reuse a generic communication object to add new features.

Constraints:

- **Model Consistency:** It is the same situation that in the previous use-case.
- **Security access:** Like in the first use-case, in order to ensure the identity of each user it is necessary to provide a log-in system.

Steps:

(a) User connects to server.

(b) User has to log in with a name and password in the repository.

(c) User makes a reference to the other communication object.

(d) User adds the new features in the communication object.

(e) User commits changes to the server.

(f) User closes the connection with the server.

3. Repository management

As it is commented in the previous use-case, there are different companies, each company works on different communication objects but in some situations they have to access to another repositories to create more complex communication objects. To allow these situations, an administrator is needed who will create the repositories and will configure the different access right for each company.

Tasks:

To create and delete repositories and configure the kind of access that each company will have.

Constraints:

- **Model Consistency:** It is the same situation that in the previous use-case.
- **Security access:** Like in the first use-case, in order to ensure the identity of each user it is necessary to provide a log-in system.

In the following a practical example of how an administrator should proceed is shown.

This first situation takes place when an administrator is going to create a remote repository.

Steps:

(a) Administrator connects to server.

(b) Administrator has to log in with a name and password.

(c) Administrator creates the new repository.

(d) Administrator sets the access rights for the companies.

(e) Administrator logs out.

The second situation occurs when an administrator is going to modify access rights to a company.

Steps:

(a) Administrator connects to server.

(b) Administrator has to log in with a name and password.

(c) Administrator modifies rights of the company.

(d) Administrator saves changes and log out.

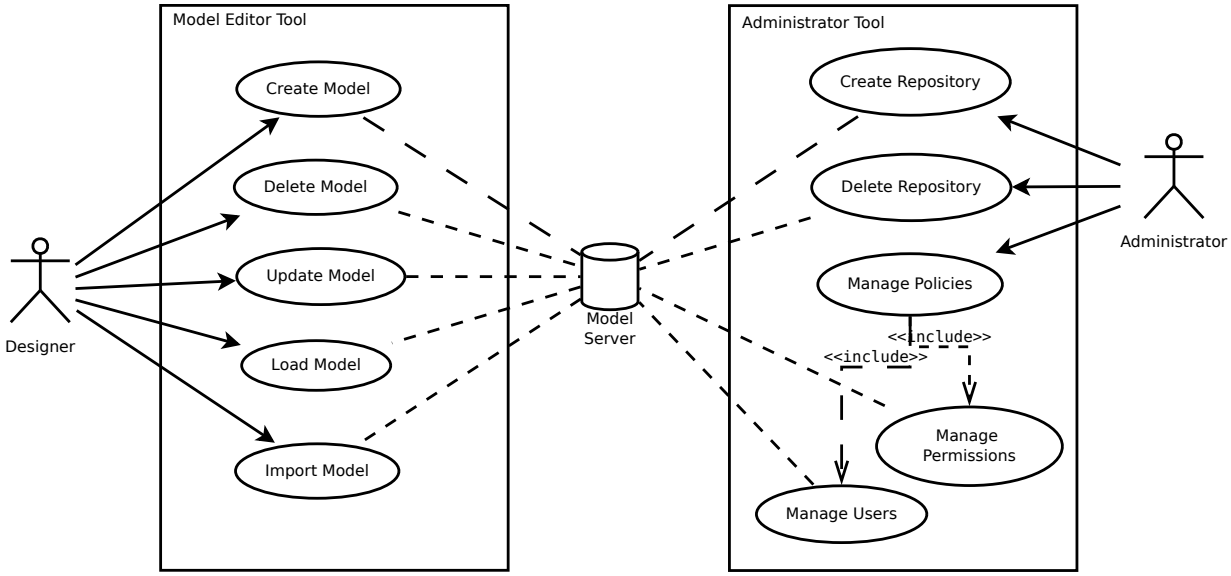

Figure 3.4: Use case diagram for managing models with a model server

Figure 3.4 summarizes the different actions that take place in the previous use-cases. The designer gather actions such as create, delete, import and load models. On the other hand the administrator can create and delete repositories and in addition manage the user rights.

Chapter 4

Method

This chapter explains how the requirements showed in the previous section are solved using CDO.

4.1 Analysis

This section synthesize different features extracted from the use-cases in the previous chapter. First defining which features are needed and second, describing which requirements to reach the features are needed.

4.1.1 Features

As have been stated in the introduction chapter, and with the previous knowledge about Communication Objects, now the specific features needed to reach the goal of this thesis are described. The features are the next ones:

1. **User Interface**: This part is the entry point for the communication object designer in the system. From the first and second use-cases a need arises to provide a user interface. This interface will support designers for working on communication object models using different actions such as to modify model remotely in the communication object repository and to download or upload models. In addition, the designer needs to be able to work with remote communication object repositories to extend models creating new complex ones.

2. **Security System**: Each use-case shows that a security system is needed. The first part is an access control to avoid the entry of unregistered users in the repository. The second part of the system is to distinguish between users who will have right to modify objects, and other users who will just have rights to use the models or reference them. Therefore, it will be necessary to have a role who will manage the different features required by the system such as to create or to delete repositories. This role is the administrator.

3. **Consistency**: From these use-cases where the objects are modified, it is a problem when these actions are happening with two or more users at the same time. One of the most important features that a system like this needs to have is the data consistency. To avoid possible problems, a mechanism to control or lock objects when one user is working on them is needed.

4.1.2 Requirements

Now that the different features have been described, it is necessary to explain the different details that are needed to cover the features.

- The first feature is the *User Interface*.
  - Model editor: it is necessary to have a tool that enable users to create/delete/modify models in the remote repository.
  - As it has been commented, it is common to extend communication objects to make new ones, with more complexity. This option has to be provided.
  - Another important capability is to allow users to import/export communication objects.
- The second feature is the *Security System*. The security system can be divided into the following aspects:
  - Security Access: This part will control which users have access to a repository. To achieve this, the next capabilities are required.
    - User authentication: Users will need a password and a user name to prove their identity. Each user has to be registered in the system.
    - In relation with the previous capability, an interesting feature for the user, would be a mechanism to change the password from the user tool or recover the password if the designer forgets it.
  - User Rights: This part will define the capabilities for the designer role. In order to support the communication object repositories, it is necessary to create user individual rights and group rights. The rights can be either read or write permissions to modify each repository. A set of group permissions make sense because a company for example will have different departments and each one with a different level of access or rights.
  - Repository Administrator: In order to manage the security system, an entity which will create user accounts, will change passwords and will set user rights is needed. The security manager needs to have a special account to log into the system. In addition, to manage the repositories, this role has to be in charge of to create and delete repositories.

- The last feature is *Consistency*: To avoid the problems explained in the previous section about consistency, it is necessary to implement some mechanism to avoid that various user can modify the same object at the same time. A locking mechanism could solve the problem.

From this analysis, it can be deduced that it is necessary to support two different roles, the model designer and the repository administrator. CDO has been selected as the most promising tool to be used in next sections.

4.2 CDO Server

This section describes features such as how to configure the server, how to solve possible problems with the server connection or how to integrate CDO with other tools.

4.2.1 Server Configuration

To start with a set of distributed model repositories the first step is to configure the server where the repositories are going to be deployed. So this section starts with the server configuration. CDO uses an XML file where all the properties have to be set (an example is shown in listing 4.1). It is possible to define several repositories in the same file and also to start repositories from different files. Some of the relevant properties to set in this file are divided into the next elements:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <cdoServer>
3
4    <!-- ================================================================= -->
5    <!-- See http://wiki.eclipse.org/CDO/Server_Configuration_Reference -->
6    <!-- ================================================================= -->
7
8    <acceptor type="tcp" listenAddr="0.0.0.0" port="2036"/>
9
10   <!-- Examples:
11       <acceptor type="ssl" listenAddr="0.0.0.0" port="2036"/>
12       <acceptor type="http"/>
13   -->
14
15   <repository name="repo1">
16
17     <property name="overrideUUID" value=""/>
18     <property name="supportingAudits" value="true"/>
19     <property name="supportingBranches" value="true"/>
20     <property name="supportingEcore" value="false"/>
21     <property name="ensureReferentialIntegrity" value="false"/>
22     <property name="allowInterruptRunningQueries" value="true"/>
23     <property name="idGenerationLocation" value="CLIENT"/> <!-- Possible values: STORE |
           CLIENT -->
24
25   <!-- <securityManager type="default" realmPath="/security"/>
26
27   -->
28
29     <store type="db">
```

```
30
31        <!-- Example http://bugs.eclipse.org/396379 (if idGenerationLocation == CLIENT)
32          <property name="idColumnLength" value="34"/>
33        -->
34
35        <!-- Period at which to execute an SQL statement to keep DB connection alive, in minutes
              -->
36        <property name="connectionKeepAlivePeriod" value="60"/>
37
38        <!-- Maximum number of store accessors (JDBC connections) to keep in the reader pool.
              The default value is 15.  -->
39        <property name="readerPoolCapacity" value="20"/>
40
41        <!-- Maximum number of store accessors (JDBC connections) to keep in the writer pool.
              The default value is 15.  -->
42        <property name="writerPoolCapacity" value="20"/>
43
44        <mappingStrategy type="horizontal"> <!-- callout -->
45          <property name="qualifiedNames" value="true"/>
46
47          <!--
48            Per default, the objectTypeCache is in-memory and contains
49            100,000 cache entries. If you want to change the size,
50            uncomment the following line and set the desired size.
51            The cache can be disabled by setting a size of 0.
52          -->
53
54          <!-- Optional:
55          <property name="objectTypeCacheSize" value="100000" />
56          -->
57
58        </mappingStrategy>
59
60        <dbAdapter name="h2"/>
61        <dataSource class="org.h2.jdbcx.JdbcDataSource"
62          URL="jdbc:h2:database/repo1"/>
63
64
65      </store>
66
67    </repository>
68
69
70  </cdoServer>
```

Listing 4.1: CDO Configuration File Example

- **Acceptor element**: This element can configure the type of connection, such as TCP, SSL, HTTP and the IP address and port to receive connections.

- **Repository element**: This element sets the different properties related to the features that the repository will support. Some examples can be to support Audits, Branches, the storage of Ecore meta-models and also to activate the security system. Another important part inside the *Repository element* is the Store element that defines the type of store factory that will be used. The Store element has four main parts.

  - **Type**: Among the different types of Store element, one can find: DBStore, Hibernate Store, DB40 Store or even a custom store. This Store Element, is an extra layer

on CDO, so it makes possible to connect many JDBC databases. There are three properties to set or leave by default, the time that a connection will be alive to execute an SQL sentence and limit of connections for writing/reading at the same time.

- **MappingStrategies**: Define the overall mapping strategy of the store element, elements settings like the size limit of elements in memory, how to handle the mapping of collection references or individual references.
- **dataSource**: this element has to match with the Store element.

Once the important parts of this file are described, the administrator, who at least initially, should be the entity who will manage the repositories is prepared to launch a CDO server. How to launch a CDO server is explained in different tutorials which can be found on the main website of CDO[1].

This XML file works well and it does not cause problems. Although on the other hand it would be desirable and it is a goal for the future in the CDO Project to manage the server configuration using a EMF Model.

4.2.2 Issues with the Server Connection

An important desirable functionality comes from the nature of CDO. Since CDO is a project based on online collaboratively work, it is necessary to ensure that in the presence of connection failures CDO can continue working or at least the damage caused by the failure can be minimized.

CDO offers three ways to solve this.

- The first case takes place when one or more designers are working with the repository and suddenly the connection gets lost. Thereby, the designer will work online while the server is on, and when the server gets offline the designer will keep working creating a branch when he commits the changes. After recovery of the connection with the repository, the designer will be able to merge this branch to update the model in the repository.

  This solution only helps up to a certain degree, because even though the designer can still work with the server, there is no real collaboration because each user will work independently from the other users and they will not be aware of changes in objects.

- The second way is to use a fail-over repository. This is a complex topology which has back-up repositories which are synchronized with the master repository. Thereby when the master fails one of the backup repositories becomes the new master. There is a separate entity, a monitor, which manages dynamically the set of repositories. In every moment the designers don't know if they are working with one repository or another, the mechanism is completely transparent. This fail-over has been tested and it works well.

[1]CDO Documentation and Tutorials `http://wiki.eclipse.org/CDO/User_Contributed_Documentation`

- The last one is to create a local clone of a repository, including full history and branches. This repository will be synchronized with the remote repository. This repository can also act like a local repository to solve some issues with the connection, for example, when there is a low latency between the remote repository and the designer.

With these mechanisms, CDO offers different ways to support connection problems.

4.2.3 Integration of CDO with Other Tools

The chapter *Related Work* showed that in CDO project there is also a project called Dawn which integrates CDO with other graphical tools like GMF and in addition it is planned to integrate CDO with Xtext.

There are other ways to integrate CDO with other tools like Xtext independently of Dawn. Although it is not part of this thesis to develop the integration of CDO with other tools, during the research of this thesis, some independent projects have been found which are able to gather Xtext with CDO. The key is the common work that these Eclipse tools have with EMF models and the easily customizable features of these tools. One of these projects is based on the modification of the Xtext component to use CDO Resource URIs. So with some modification it is possible to save some models in the CDO repository and see the Xtext code of this models directly.

4.3 Administrator Role

This section describes in detail which features are needed or would be desirable for this role. This role manages the access control and also the user rights for editing/creating/deleting models. In addition, this role has to manage the repositories.

4.3.1 Security Management

The first point to study is the security. As has been commented in the previous section for the security system it is necessary to cover features such as an access control for users and a permission system to limit different roles (e.g. like different departments in a company).

To reach these goals, CDO has an Ecore model implemented on the server side to provide the security. Figure 4.1 shows the model. The model has the realm as root element which contains the following elements:

- Role: describes the functions and their access rights.
- User: describes the access to the repositories. Users can have different roles and belong to different groups.
- Group: gathers users who share the same rights. Groups can have many roles.

[2]CDO Security Manager Model: `http://wiki.eclipse.org/CDO/Security_Manager` last visited: 4/26/2013

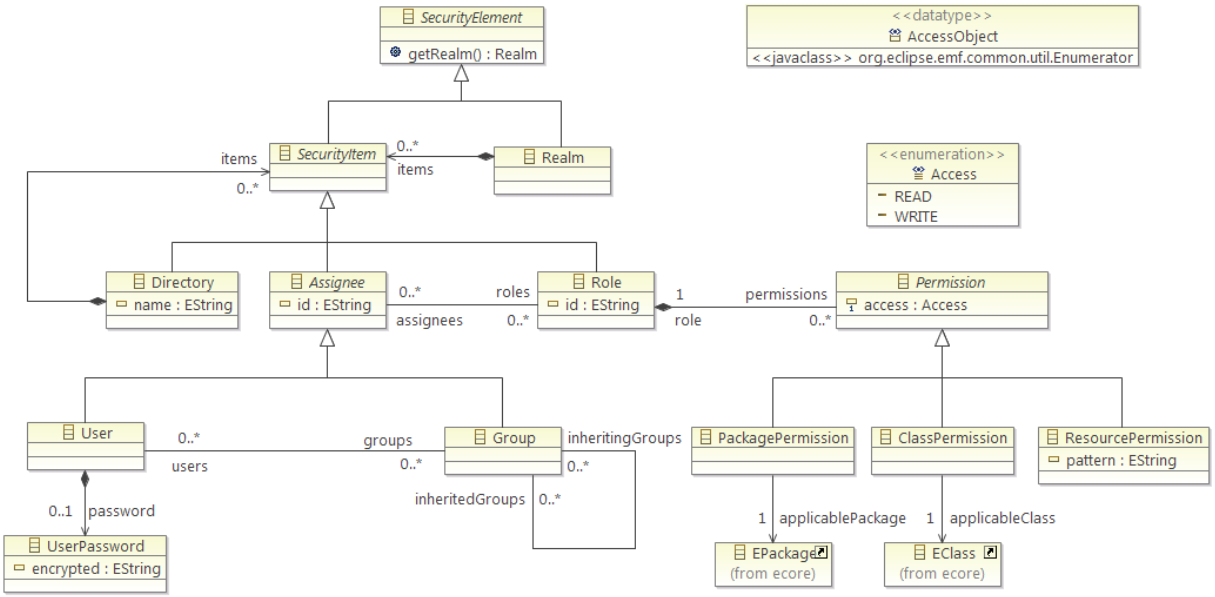

Figure 4.1: CDO default security model[2]

- Directory: gathers any number of these four elements. For the purpose of organizing the elements.

There are three different types of roles: NONE, READ and WRITE. These roles can be assigned to the following elements:

- ResourcePermission: specifies access right for any resource or folder.
- PackagePermission: specifies access right for all EClass included in this Package.
- ClassPermission: specifies access right for the EClass.

In order to activate this feature in CDO, it is necessary to add the line:

```
<securityManager type="default" realmPath="/security"/>
```

in the configuration file in the *Repository Element* and to restart the server. This configuration file has been explained in the previous subsection. The administrator will have to log in with the administrator password. By default the user name is “Administrator” and the password is “0000”.

When the administrator has logged in into the repository, after opening a transaction, the administrator will have access to the Security Realm. The administrator will be able to add new role profiles and set what kind of access this role will have to the different parts in the repository,

[3]CDO Security Manager Interface: `http://wiki.eclipse.org/CDO/Security_Manager` last visited: 4/26/2013

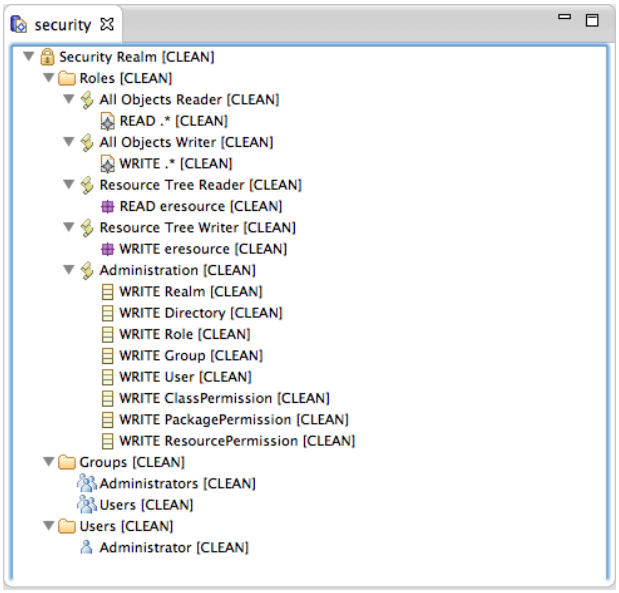

Figure 4.2: A view of the security realm interface[3]

as shown in the figure 4.2 with the Administration role. The "Groups" branch is used to define user groups. In this part, the administrator will add users to each group and assign roles to the group. The last part, is where the administrator will add/delete users. The administrator also will change the password of the users or apply individual roles for each user.

From this security system model that CDO uses, there is a good base to implement one of the goals of this thesis namely to have an access system and a permission system to manage the users of the repository. Although it can be considered a beta version because currently it is not working completely. For example the WRITE access is only checked at commit time so the designer can create or modify objects with the editor but the changes will be rejected only when the user try to commit. Another aspect that doesn't work correctly is that when there is a change in the security realm, these changes will not have any effect until the next restart of the server. These aspects and others more are addressed in a Bugzilla list of CDO where the developers work continuously.

An independent solution from CDO could be to create an external database of users which can manage different groups and set the different kind of permissions. Then modifying the CDO API code related to the access in the repository it should be possible to introduce the necessary SQL sentences to ask to the different users for their user-name and password, and when one designer is going to access to a repository, to check what kind of access the user has and then to show the content of the repository.

4.3.2 Repository Management

The second main objective of this tool is the management of repositories. CDO does not offer an administrator tool where one can add or delete repositories directly. In CDO the way to start

a repository is by using the XML configuration file. In this file one can set the different features as has been explained in the previous section. One server can have one or more repositories. CDO offers a functionality through the OSGi console to be able to add, delete repositories. This console can only be used from the server side. On the other hand, CDO provides an API (org.eclipse.emf.cdo.common.admin) to manage repositories remotely. Other option could be to exploit the API provided by CDO and integrate these functions with a graphical user interface to ease the repository management. Due to time reasons this interface is not yet evaluated.

Another possible solution could be to create an administrator tool with an interface where the administrator can set the different features that the repository is going to have. Afterwards this interface can generate the XML file, where there is a default schema and the different fields are filled with the features that the administrator has chosen with the graphical interface.

4.4 Designer Role

This section describes in more detail which features are needed or would be desirable for this tool. This tool provides all utilities that a designer will need to work with a repository like the possibility to create/delete/modify communication objects.

4.4.1 User Interface

As has been explained in the previous chapter, a user interface is needed to ease the designer work.

The user interface in CDO is based on Sessions and Views which offer the access to the different features of the server. The first one is the CDO Session view. It is the main access point of a client to a server repository. A designer can open an arbitrary number of sessions but each session maintain a revision cache, so it is not recommendable for the expensive use of memory. From each session a designer can open several CDO Views. In a sense, the CDO Views are light weight entities in comparison with a CDO Session.

There are three main views in CDO:

- **Transaction**: It is the only read-write view which CDO offers. From the transaction a designer can create resources and then open the CDO Editor to start working with the models.
- **Read-only View**: It is a read only access to the resource that shows the latest state of the repository.
- **Audit view**: It is a special read only view that allows to look at an old state of the models.

The CDO Editor allows designers to create models, or to add root objects from the packages which are registered. To register a model in CDO it is necessary to generate the Java classes of the model with a .genmodel as was explained in the Chapter 2: *Related Work*. For CDO there is a special .genmodel which prepares the Java code for CDO. During the process of creation of

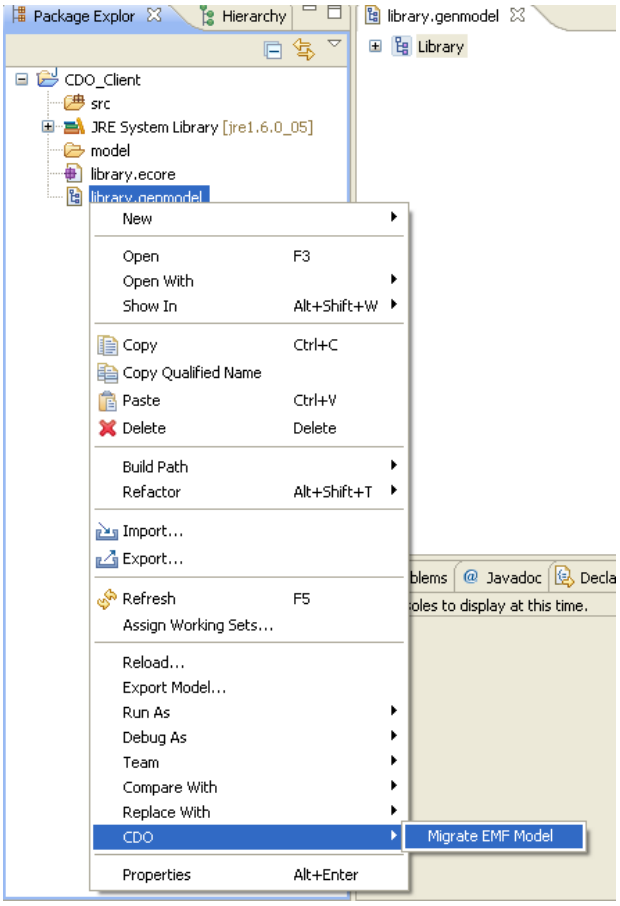

Figure 4.3: Migration of generator model (.genmodel) to CDO

the .genmodel the designer has to choose “Ecore Model (CDO Native)” . After that it is possible either to import a created model in the repository and work with it or add the required plug-in in the Eclipse Run Configuration of the client application.

There is a second option if the model already has a .genmodel, then to migrate it to CDO. This option is very simple, the figure 4.3 shows the option in the contextual menu which allows to do the migration. In addition, the designer will be able to import models from other projects in the workspace or the filesystem and to export the models which are in that moment in the repository.

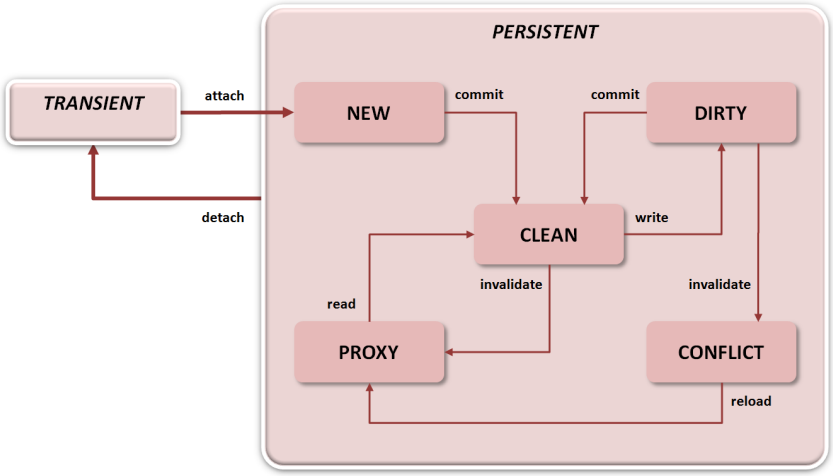

Figure 4.4: CDO State Machine[4]

While a designer is working on a model, he will have information about the state of the objects. These states are:

- **New**: When a designer creates a new object, the object will have this state.
- **Clean**: Once the designer commits the changes, the state of the object will be “Clean”.
- **Dirty**: When a designer modifies an object, the state will be “dirty” before to commit changes.
- **Conflict**: When two or more designers are working on the same model, for example user A modifies an object, meanwhile user B, modifies the same object and commit the change. In this case, user A will be notified, and the state of the object will be “Conflict”.
- **Proxy**: It is an intermediate state. When the object is in “Conflict” state, if the state is reloaded, the object will pass to “Proxy”.

[4]CDO State Machine taken form the blog of an CDO core developer: `http://thegordian.blogspot.de`

Figure 4.4 shows the state machine of the states previously described.

Using the capabilities of Eclipse Client Platform, it is possible to use this framework for building applications where the different views can be integrated. This way it is possible to create a customized interface that gather the necessities of the company.

4.4.2 Consistency Mechanism

One of the most important features that a system of distributed repositories needs to have is consistency. Many users can work on different parts of a model and commit changes at the same time. CDO only accepts the first commit, so the second designer should receive a notification before to commit any changes. To avoid possible conflicts during a modification on an object CDO provides an explicit mechanism of locking objects. Therefore a designer can use this feature to lock one or more objects at once. The nested objects can be affected as well. Other designers will not be able to make changes in these objects until the locks are released. The objects will be unlocked automatically when the transaction is closed.

The default functionality of this feature is simple. When a designer locks an object or part of it, the rest of designers do not know that the object is blocked. When these other designers try to modify the object, at commit time, they will receive a message telling that they can not do modifications. Like a first approach, this feature achieve the goal, the other users can not modify the objects while the object is blocked. But it is not a comfortable behaviour. The optimal situation would be that the other designers are informed about that the object is blocked, with a message, or some kind of information in the CDO editor. It is possible to change this behaviour with the modification of the Java code of the classes that implement this feature. These Java code can be found in IStoreAccessor.DurableLocking in the *org.eclipse.emf.cdo.server* package in the server side and CDOLock in the package *org.eclipse.emf.cdo* package for the client side.

4.4.3 Download/Upload Communication Objects

As has been shown in the first use-case from the previous chapter it is necessary to have a mechanism to download/upload communication objects in the repository.

CDO allows to the designers export/import resources easily. In the case of exporting a resource, the designer just has to select the resource that he wants to export and select where to download the model.

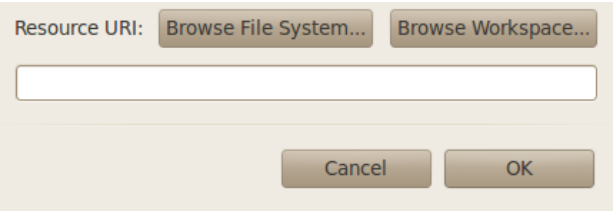

Figure 4.5: Export resource window.

The opposite case is very similar. In this case the designer will need to select the import option in the transaction view. Then, after providing the URI where the resource is located, the designer just needs to commit the changes. It is important to remember that the generator model (.genmodel) has to be migrated to work with CDO.

Another option that CDO offers is "Load Resource". This option it necessary when one designer wants to work with communication objects in another repository. As it can be seen in figure 4.5 the designer will have to introduce the URI of the resource where the communication object is. To make reference to resources in another repository the designer has to introduce the URI like that: "cdo.net4j.tcp://IP:PORT/repository_name/resource_name" where IP is obviously the IP address of the repository and PORT is the port number. Again this features can be integrated in a customizable interface.

4.4.4 Access to the Repository

This subsection shows the security system from the side of a designer. To access to the repository in CDO, the designer has to justify that he is a registered user. In CDO the designer will have to connect to the server and then open a new session. After that, CDO will ask for the User ID and password. Previously the administrator must have created this User account. By default, CDO does not offer any features to allow the designer to change his own password or a mechanism to ask the administrator for a new password from the user interface. This would be a good feature to have. It could be implemented as a functionality in the designer tool interface, more specifically in the CDO Session view, to allow designers to change their own password introducing first the old password. On the other hand, it would be also necessary to add this feature in the security model, to allow the Administrator to change user password.

4.4.5 Versioning

About model versioning, the *Related Work* chapter does not show anything about this feature in CDO. Nevertheless CDO implements the Audit View which can offer a way to get versioning. Since it is possible with this CDO View to see previous versions of the models in the repository, it would be possible to use a previous version if it is necessary. Although currently there is no way to make a direct "rollback" with this view, it is possible to save or export an old state of the model in an XML file and load the model if it is necessary.

Chapter 5

Experiments and Results

This chapter summarizes the features that have been analyzed in previous chapter showing what level of completion have been reached.

As the chapter *Method* has shown, CDO offers a good base to build applications with a set of remote repositories to work collaboratively, although CDO has many features which are still under heavy development . First, CDO covers the main desirable features for the robotic case. Some of these features can be still considered like a "beta" version, because the default solution from CDO is not completely developed. Other features work with a behaviour that in the beginning can be sufficient. One of the advantages of CDO is that it is an open source project and this allows to adapt the different features to better fit with a concrete behaviour. Also it is possible to get these "green" functionalities and to implement a personal solution. As most of Eclipse projects, there is a good community support.

5.1 Analysis

This section shows up to which level the needed features are covered with CDO.

- **Server Configuration**: Although it would be better to use a model for the configuration, it is not a big deal to use an XML file because it is possible to prepare a script with a template to generate the file with the specific properties.

- **Issues with the Server Connection**: About the different options that CDO offers to solve the possible connection problems, the mechanism that works best is the fail-over repository. It has been tested without major problems. The offline-branch mechanism worked well at first, when only a couple of users were working and they needed to merge just once. However, after trying different iteration turning off the repository, some problems were found when the state of the objects are not uploaded. At first it could work like another layer to avoid connection mistakes but it requires further investigations for a proper use of the feature.

- **Integration of CDO with Other Tools**: It has not been tested because it is not relevant in this thesis but it has been shown in previous chapters that there are different ways to do it by e.g. using CDO Dawn or developing an own solution.

- **Security Management**: Regarding security in CDO. The Security Manager Model has been tested. The test showed that this model is still a beta. It has a good potential because the model covers almost all features that are desirable for the cases in this thesis. Nevertheless it has many missing points that do not allow to use a default version to see at least how the different parts of the model work.

- **Repository Management**: The repository management is a feature that CDO provides but it is not yet matured. The effort in this part should be to develop a graphical interface which can gather the different actions to ease the administrator task.

- **User Interface**: CDO implements a complete system of views where the different features about the designer actions take place. An important part is that a designer can work directly with the default system that CDO provides. The other part has been commented in the previous chapter. All these views can be included in a customized interface.

- **Download/Upload Communication objects**: This feature can be considered as part of the user interface. This means that would be necessary to implement these features in the designer interface because the current way to work with these features is uncomfortable.

- **Access to the repository**: As it has been commented about the security system, using the Security Manager model, it needs further improvement to have a good working feature.

- **Versioning**: This is a good feature. To have a history of the models allow to access to different old versions. This is important if it is necessary to do a rollback or work in a parallel version of one model.

Chapter 6

Conclusion and Future Work

This thesis addressed the problem of collaborative work on distributed communication object repositories. To summarize, this document showed that it is possible to work collaboratively on communication object repositories using CDO. It started with the analysis of the problems and difficulties that are involved in this kind of work. For that the chapter *Introduction* showed a set of use-cases and from these use-cases a first overall idea of the different challenges was presented. After that, in chapter *Related Work* a basic knowledge about modeling was presented and also an introduction of two different tools which allow to collaboratively work on distributed repositories. In addition, the two tools were compared and CDO was finally chosen. Afterward the next chapter *The Robotic Case Study* introduced the specific focus on communication objects where the use-cases were analyzed again and the features were extracted with the specific focus on communication objects. Chapter *Method* showed how CDO can cover the different features. Chapter *Results* showed now far the goals have been solved.

CDO is a tool in constant development that improve the different features, add new ones and there are many groups using CDO as base for their own specific purpose. This is an important point. An open source project allows to extend features easily and to integrate CDO in other systems. Although this thesis only defines a solution in a conceptual way, it shows that it is possible to start collaborative work on distributed repositories. There are many features under development, it means that it is possible to start working with CDO and build an independent solution of the features that are not completely developed, but it is also possible that half year later some of these features are working completely.

6.1 Future Work

This thesis is an entry point to start with the implementation of a distributed system of repositories. The very first step has to be the implementation of a basic architecture that can consist of one repository and a group of designers. With this simple example it is possible to test the Security Manager, setting different user rights and adapt the security system to a concrete purpose. Another important part to test, is the user interface where different designers can start working with models in the repository and try the different features such as create, delete models and to see if

it is necessary to adapt the behaviour of these functionalities. Once this basic architecture works correctly one can add more complexity creating different groups with different rights, adding a second repository and try to create nested communication object using references to these different repositories. On the other hand, regarding designer and administrator interface, there is a new Eclipse project called EMF Client Platform which allows to integrate different EMF technologies such as CDO or EMF Store for building applications. It would be very interesting to study this project for building CDO applications for the designer and administrator interface. With these experiments, one can further refine the requirements and add further solutions to meet them.

Bibliography

[FB03] Ed Merks Raymond Ellersick Timothy J. Grose Frank Budinsky, DAvid Steinberg. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley, 2003.

[Gro09] Richard C Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[mof05] *Meta Object Facility (MOF) Specification Version 1.4.1*. Number formal/05-05-05. Object Management Group, 2005.

[omg03] *MDA Guide Version 1.0.1*. Object Management Group, 2003.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[SSL12] Christian Schlegel, Andreas Steck, and Alex Lotz. *Robotic Systems - Applications, Control and Programming*, chapter Robotic Software Systems: From Code-Driven to Model-Driven Software Development, pages 473–502. InTech, 2012. ISBN 978-953-307-941-7.

[uml11] *OMG Unified Modeling LanguageTM (OMG UML), Superstructure Version 2.4.1*. Number formal/2011-08-06. Object Management Group, 2011.