Conversion of Octopus UML Models into Eclipse UML2 Models

Student Project

Submitted by:
A. Jibran Shidqie
ajibran.shidqie@tu-harburg.de
Information and Media Technologies
Matriculation Number: 27199

Supervised by:

Prof. Dr. Ralf Möller
STS - TUHH

M.Sc. Miguel Garcia
STS - TUHH

Hamburg, Germany
August 2006

Declaration

I declare that:
this work has been prepared by myself,
all literally or content-related quotations from other sources are clearly pointed out,
and no other sources or aids than the ones that are declared are used.

Hamburg, 09.08.2006

A. Jibran Shidqie

Abstract

Nowadays Model Driven Architecture (MDA) has become more popular within many organizations as an approach to design their application. The flexibility and the reusability of the design makes them even more important. Unified modeling language (UML) which recently known as primary modeling notation plays a central role in this architecture. OCL Tool for Precise UML Specifications (Octopus) and Eclipse Modeling Framework (EMF) are the examples of these UML tools which have their own strength.

Octopus has focused so far on modeling-time checking of well-formedness of UML + OCL models, and their IDE support. In contrast, the Eclipse modeling platforms has focused less on compile-time correctness of OCL models as on providing graphical tooling to manipulate instantiations of domain-specific languages defined by the modeler. The combination of their strength will demonstrate powerful capabilities of designing model in application development. This student project will combine the strong point of both tools by providing the bridge between them. The converter which transform the Octopus Model to EMF based Model will be the concrete result of this work.

Table of Contents

Table Of Figures

I Introduction

I.1 Background

The development of enterprise-level software systems requires applying appropriate techniques from Software Engineering in order to help software architects to provide solutions. Several challenges, e.g. reusability and unplanned system evolution, have to be coped with. Model Driven Architecture (MDA) comes into play to help the software architect to achieve all of those needs.

In MDA, as its name suggests, the model plays as central role. There are many reasons why models become so important. Models represent abstractions of real-world systems that can be used in communicating intricate ideas. People can understand faster and easier the model than the programming code. Early experiences consisted in using the notation and tools of the Unified Modeling Language (UML) to document system perspectives, which can be mapped into design documents and into programming language code. Initially the last transformation was performed manually, but today as we can see there are tools that can generate code automatically.

This Student Project introduces two such tools, OCL Tool for Precise UML Specifications (Octopus) [7] and Eclipse Modeling Framework (EMF) [3]. Both of them have their own strengths which can be useful under different situations. Octopus has focused so far on modeling-time checking of well-formedness of UML together with its OCL models, and their IDE support. On the other hand, EMF has focused on providing graphical tooling in order to manipulate the instantiations of domain specific languages defined by the modeler.

I.2 Objectives

We can combine the strengths of both tools to maximize benefits. Octopus provides a very good way on modeling time by ensuring the OCL expression are syntactically correct. And EMF on the other side has a very useful features around generating code and other artefacts (GUIs). When modeling with EMF, we have to wait

until runtime in order to know whether syntax errors are contained in defining OCL expression. Preparing the model in Octopus can avoid such errors and guarantee runtime executability. Bridging between these two powerful tools would be a very interesting works, and might be a good solution generally to MDA which keeps evolving.

This Student Project is a continuation of previous work, a converter from an Octopus class model to an EMF model. At the beginning of this StA, that plugin (octopus2emfatic) could convert the octopus model into Emfatic form [4] without OCL constraints. By building upon that plugin, this StA consists of two related tasks:

1. Conversion of Octopus UML models to Eclipse UML2 Abstract Syntax Tree representation (AST) [17].
2. Conversion of Octopus UML + OCL models to the Emfatic file format.

The conversion of Octopus models to UML2 ASTs aims at reusing the tooling for graphical representation and manipulation of UML2 models. UML2 as an Eclipse project provides the interoperability to other EMF based components and (commercial) tools, making easier to use those tools to extend the functionality of our original models.

The emfatic format, which represents textual representation of the Ecore model (EMF model) is a human-readable textual notation for EMF models. The second task aims at completing the conversion process in our toolchain by including OCL constraints.

In summary, the overall objective of these tasks is to contribute to a toolchain for the model-driven development of enterprise-class systems.

I.3 Organization Document

Chapter I gives some explanation about the background and the objectives of this project. This introductory chapter also provides a brief overview of the terms used in the rest of the report.

Chapter II explains about the conversion process from Octopus model into UML2 model. It provides general information about what constitutes the Octopus model and the UML2 model.

Chapter III focuses on how to add the constraints to complete the conversion process from Octopus model to UML2 model. It also discusses the limitation and the alternatives of the solution.

Chapter IV describes how to ensure that the model is kept up to date with the source form of the model. Information about the validation process against the defined constraints is provided here.

The last chapter presents the summary and briefy discusses the future of this project.

I.4 Preliminaries

I.4.1 Object Constraint Language (OCL)

The Object Constraint Language (OCL) [6] is a language that enables describing expressions and constraints on object-oriented models and other object modeling artifacts. An expression is an indication or specification of a value. A constraint is a restriction on one or more values of (part of) an object-oriented model or system. For more detail information, please refer to [8].

I.4.2 Octopus

Octopus stands for "OCL Tool for Precise UML Specifications". The information in this subsection is summarized from [7]:

Octopus offers two important functionalities:

1. *Octopus is able to statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.*
2. *Octopus is able to transform the UML model, including the OCL expressions, into Java code.*

Octopus fully conforms to version 2.0 of the OCL standard. All new constructs, like derivation rules and initial value specifications, are completely supported. Furthermore, Octopus offers the possibility to view expressions in an SQL-like syntax. The semantics of the original expressions,

written in the standard syntax, remain fully intact, while their appearance become more familiar for those who have been working with databases.

Octopus is able to generate a complete 3-tier prototype application from your UML/OCL model. The middle tier consists of plain old Java objects (POJOs). These POJOs include code for checking invariants and multiplicities from the model. OCL expressions that define the body of an operation are transformed into the body of the corresponding Java method. Derivation rules and initial value specifications are transformed as expected.

The storage tier consists of an XML reader and writer dedicated to the UML/OCL model. It stores any data content in the prototype application in an XML file. Naturally, it is also able to read the content of this XML file into your prototype application. Furthermore, the application may be regenerated, for instance, because one of the classes was missing an attribute, and the reader will still be able to read the XML file. The reader will read the contents of the XML file and produce objects for whatever classes, attributes, and association ends are still in your model. New model elements simply remain empty.

The user interface tier consists of an implementation of a plug-in for the Eclipse Rich Client Platform (RCP). From a Navigator view that shows you all instances in the system, you are able to create and examine instances of your UML/OCL model. Of course, the invariants or multiplicities of an instance can be checked by pushing a single button.

I.4.3 EMF & Emfatic

The Eclipse Modeling Framework (EMF) is an open source framework for developing model-driven applications. It creates Java code for graphically editing, manipulating, reading, and serializing data based on a model specified in XML Schema, UML, or annotated Java [1]. In addition to that, EMF also provides runtime support for the models which covers change notification, persistence support, and reflective API for manipulating EMF objects generally.

The core model of EMF called Ecore which describes application data models and knows as Ecore meta-model.

Figure I-1 shows the complete class hierarchy of the Ecore model. Figure I-3 shows the dependency diagram of Ecore metamodel.

To generate the model into code, EMF has generator model called genmodel, which is a wrapper of the ecore model that provides options that can be configured relevant only to the code generator, such as what packages to use, and where the generated code should go.

Emfatic is a language designed to represent EMF Ecore models in a textual form. It can be useful tool for viewing or building the Ecore models. Both of this form can be derived from each other. Emfatic provides the generator which converts Emfatic to Ecore. and vice versa.

Figure I-2 shows the class hierarchy of Emfatic metamodel.

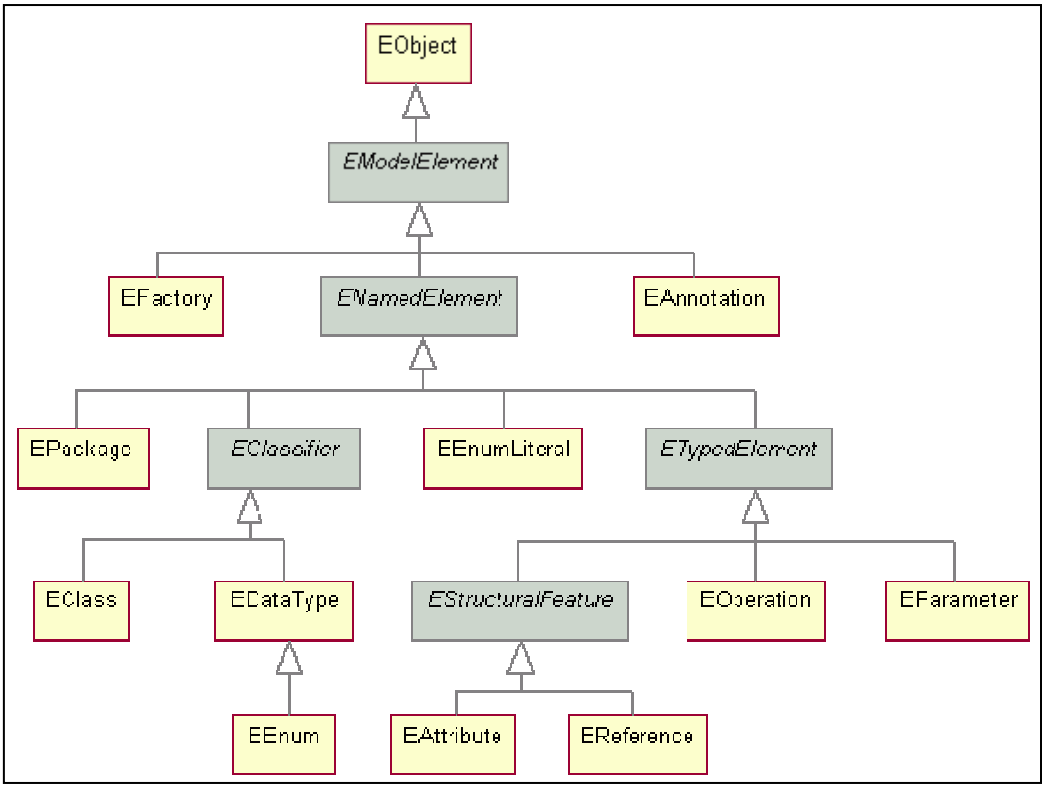

Figure I-1. Class Hierarchy of an Ecore Metamodel [11]

I.4.4 UML2

The UML2 project (an Eclipse Tools sub-project) is an EMF-based implementation of the UML 2.x metamodel for the Eclipse platform [17]. This implementation is provided to support the development of the modeling tools.

UML2 in their implementation has several goals:

1. Provide a common XMI schema to facilitate interchange of semantic models.
2. Provide test cases as a means of validating the specification and implementation.
3. Provide validation rules as a means of defining and enforcing levels of compliance

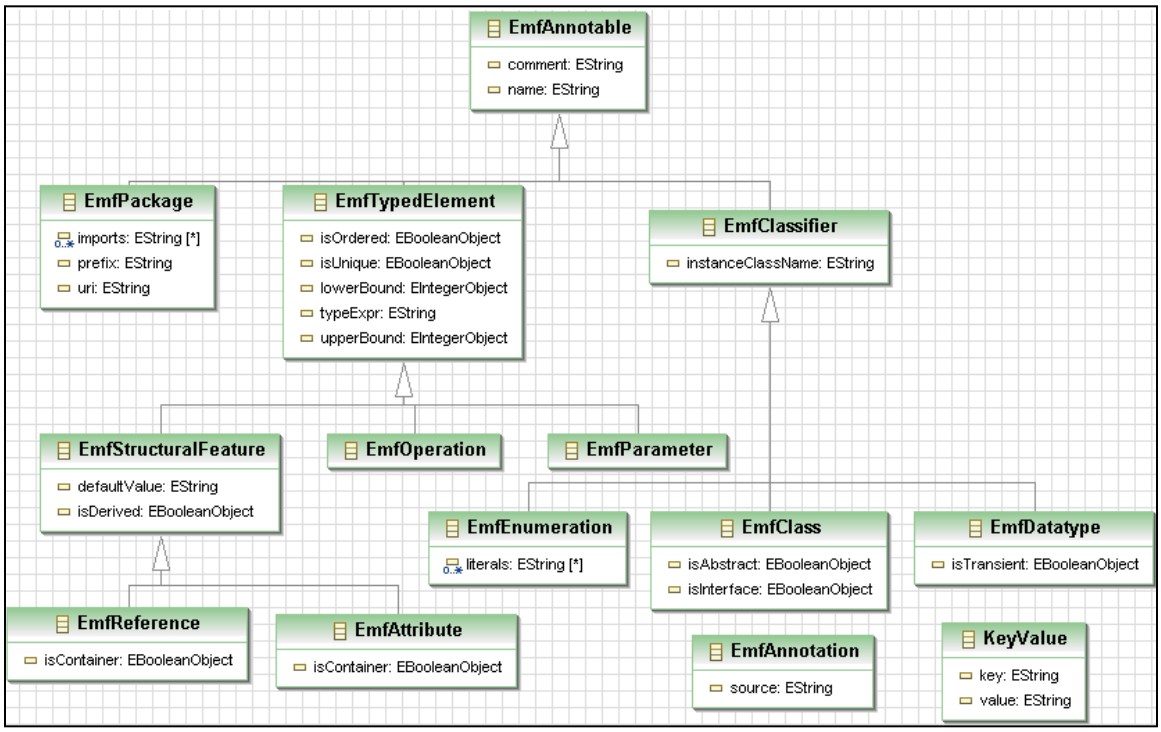

Figure I-2. Class Hierarchy of Emfatic Metamodel

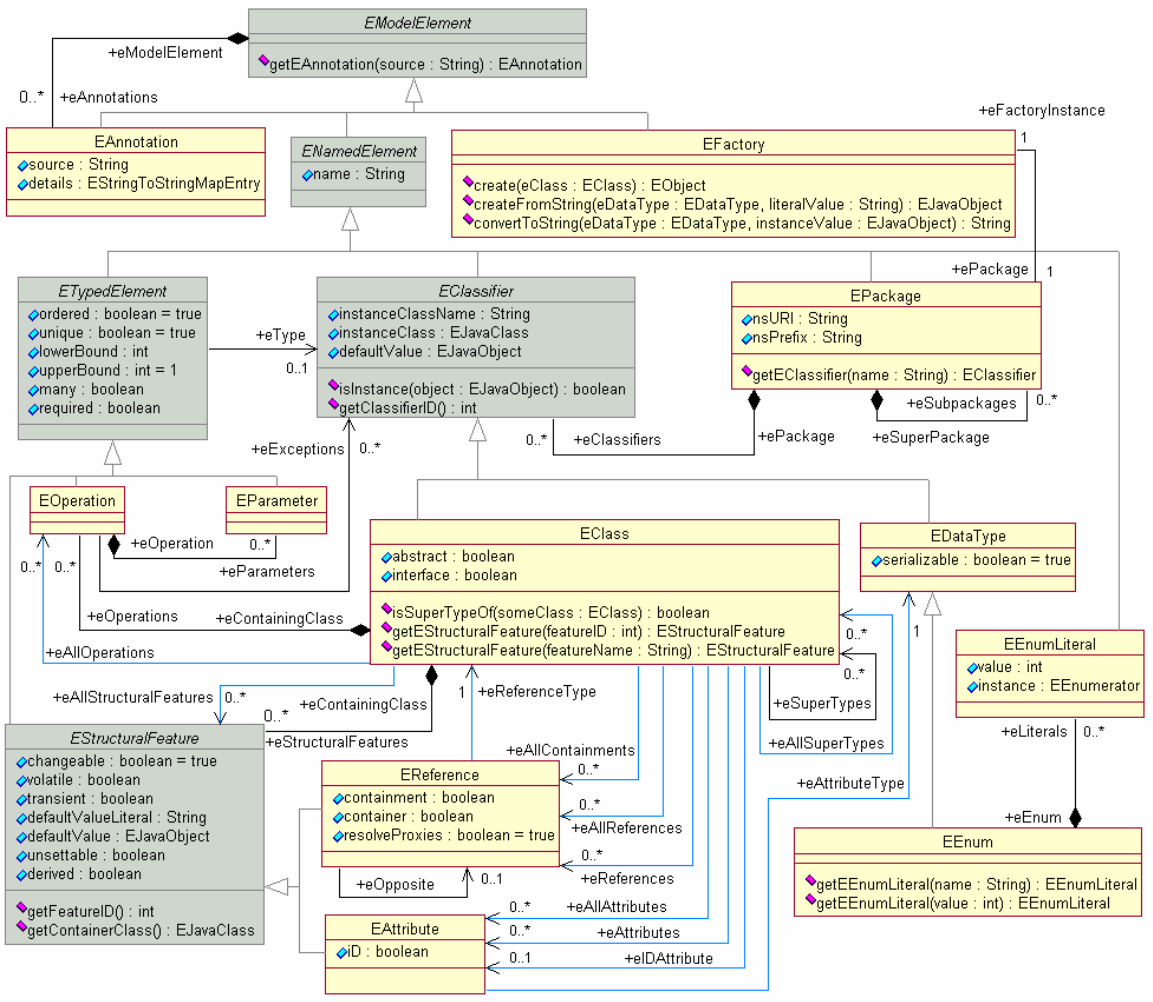

Figure I-3. Class model of Ecore Metamodel

I.5 Summary

MDA gives the flexibility and the ease to the software architect to build such a big-scale application. UML plays an important role as the model of MDA. Octopus is one of the UML tools which can express UML class models and OCL constraints.

Furthermore, Octopus can guarantee the well-formedness of that model at modeling time. EMF and UML2 is an Eclipse based framework which gives the possibility to generate the code from the structured data model. It provides the editor in which we can manipulate the model. The objective of this student project is to extend the Octopus modeling platform to convert Octopus models to UML2 Model, in order to combine all the benefits both of Octopus and EMF.

II Octopus to UML2

II.1 Octopus

To get the full functionality of octopus, it has to be installed as eclipse plugins. The source code can be found at http://sourceforge.net/projects/octopus/. After having it as eclipse plugins, we can create an octopus project. The functionality provided in Octopus is available only on projects with an Octopus nature.

The Octopus nature can be achieved in several ways. It can be added to Java, Plug-in, and simple projects. To create a simple project with the Octopus nature, the wizard will help you, 'New>Project>Octopus>NewOctopusProject' and follow the directions. To add the Octopus nature to a Java or Plug-in project, create the project in the normal way. Next, either use the 'Octopus' context menu on the project and select 'Add Octopus Nature', or use the button with the red 'O'. Another way to add the Octopus nature is to select the 'Add Octopus Nature' entry in the Octopus menu.

This nature will then specify two special folders, 'model' and 'expression' by default. The first is the folder where the UML elements are stored (*.uml), and the latter is the folder where the files that contain OCL expressions are stored (*.ocl). All These folder are configurable in the project properties. Combination of these two elements constitutes as Octopus models.

II.1.1 Royal and Loyal Example

Octopus provides several example projects in their site (http://www.klasse.nl/ octopus). One of them, Royal and Loyal, is used within this work as the example case.

Royal and Loyal (R&L) models the computer system of a fictional company. It handles loyalty programs for companies that offer their customers various kinds of bonuses. Often, the extras take the form of bonus points or air miles, but other bonuses are possible as well: reduced rates, a larger rental car for the same price as a standard rental car, extra or better service on an airline, and so on. Anything a company is willing to offer can be a service rendered in a loyalty program [8]. Figure II-1 shows its UML class model. Like other UML model, it shows no dependency on whatever programming

language that will be used to build the system.

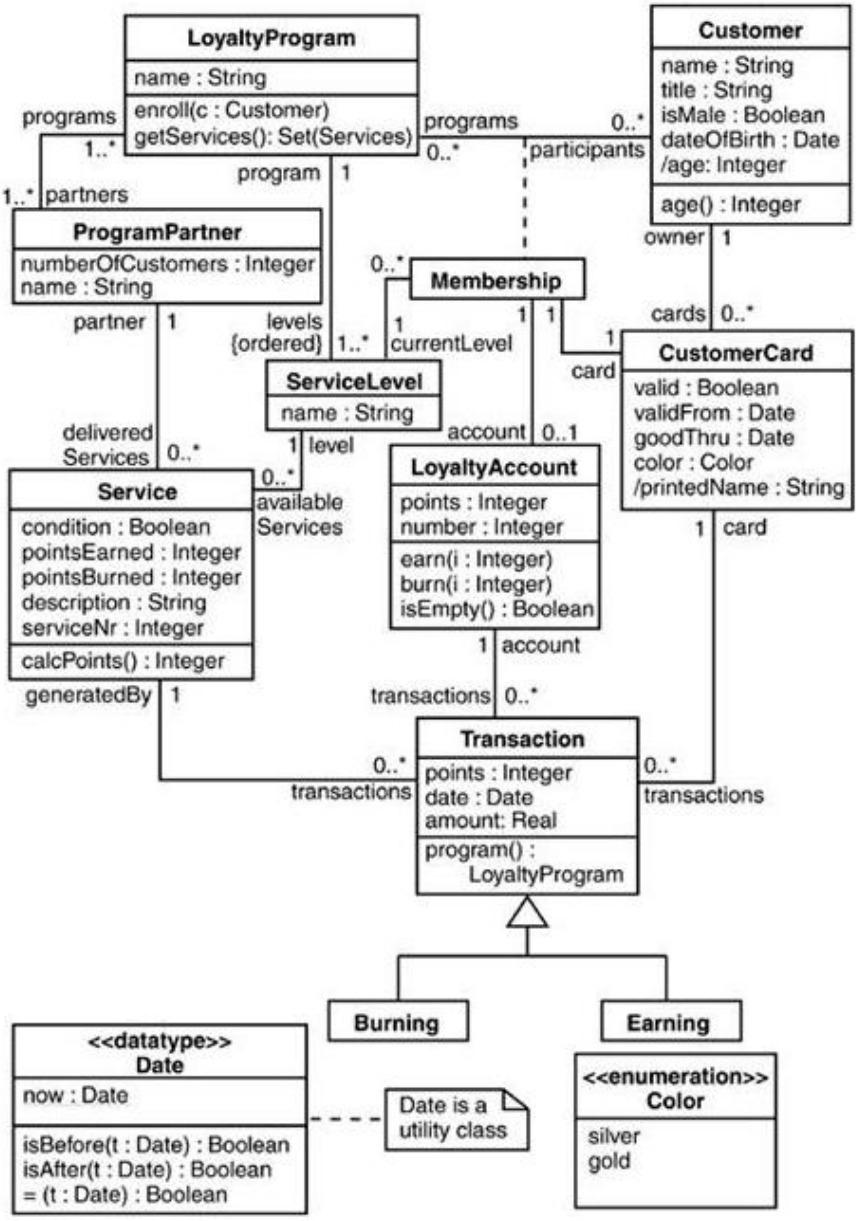

Figure II-1The Royal and Loyal Model [8]

II.1.2 Octopus UML model

The octopus UML model can be obtained in two ways:

1. The model can be manually created within Octopus itself using a textual representation of UML,
2. The model can be imported from the XMI format. This XMI format normally can be derived from the UML modeling tool.

Octopus has provided the import functionality to make things easier for octopus user. The result of this import action is the textual representation of UML. This functionality created based on XSLT, but still have some problem caused by the dialect differences of XMI format of various modeling tools. The XMI format of Royal and Loyal is already given from the example, so we can directly import it into the octopus UML model.

Figure II-2 shows part of the model in the textual representation.

```
<package> RandL
 + <class> Burning <specializes> Transaction
<endclass>
 + <class> Customer
<attributes>
 + name: String;
 + title: String;
 + isMale: Boolean;
 + dateOfBirth: Date;
 + /age: Integer;
 + gender: Gender;
<operations>
 + age(): Integer;
 + birthdayHappens();
<endclass>
.
.
.
<endpackage>
```

Figure II-2. Example of Royal and Loyal UML model - RandL.uml

II.1.3 Octopus OCL expression

OCL expression are stored in the file with the context name as the filename and '.ocl' as the extension, for example Customer.ocl. Each OCL file needs to be structured according to the rules in the OCL specification version 2.0. All OCL files should be placed under the dedicated expression folder in the Octopus project.

```
package RandL

context Customer
  inv ofAge: age >= 18

context Customer::birthdayHappens()
  post: age = age@pre + 1

context Customer
  def: wellUsedCards : Set( CustomerCard )
    = cards->select( transactions.points->sum() > 10000 )
  def: loyalToCompanies : Bag( ProgramPartner )
    = programs.partners
  def: cardsForProgram(p: LoyaltyProgram) : Sequence(CustomerCard)
    = p.Membership.card
.
.
.
endpackage
```

Figure II-3. Example of Royal and Loyal OCL Expression – Customer.ocl

Figure II-4. Octopus UML model of Royal and Loyal

All OCL expression which exist in the octopus project (project with the octopus nature) will be checked whenever the project is rebuilt. First it will check the UML files, then if there are no errors found, all OCL files are checked, otherwise no OCL files are checked.

Figure II-3 shows part of the ocl expression of the Customer context. Figure II-4 shows the Octopus UML model of Royal and Loyal in Abstract Syntax Tree (AST) view.

II.2 Conversion from Octopus to UML2

II.2.1 UML2 model

A model contains three major categories of elements: Classifiers, events, and behaviors. Each major category models individuals in an incarnation of the system being modeled. A classifier describes a set of objects; an object is an individual thing with a state and relationships to other objects. An event describes a set of possible occurrences; an occurrence is something that happens that has some consequence within the system. A behavior describes a set of possible executions; an execution is the performance of an algorithm according to a set of rules [16].

The detail specifications about UML2 metamodel can be found in [15] and [16]. Those are the official document released by Object Management Group (OMG, http://www.uml.org). The detail information about how to get the source code and how to install it properly in order to get it run in Eclipse can be found at http://www.eclipse.org/uml2.

II.2.2 The conversion

Octopus has implemented visitor pattern in modeling their AST. This makes the conversion process much easier. What we should do is to provide the 'visitor' which will visit all Octopus elements according to Octopus's visitor pattern implementation.

This conversion action uses a number of visitors to visit the input Octopus model (UML model and OCL expression). All of these visitors implement the package visitor of Octopus (nl.klasse.octopus.modelVisitors.IPackageVisitor) which is already provided by

Octopus. An implementation of the IPackageVisitor can specify the visit to go through the Classes, Associations, Operations, etc. by defining the operations ‘visitXXX()’. If they return true the corresponding items are visited, otherwise they are not. Each visitor visits the input based on the corresponding item that they visited and then translates them into UML2 model structure.

The basic idea of the visitor used in this conversion is taken from the visitor of octopus2emfatic plugin which converts Octopus model into Emfatic. The visitor defines several maps. Each map maps an Octopus element to UML2 element, for example, Octopus package (nl.klasse.octopus.model.IPackage) to UML2 package (org.eclipse. uml2.uml.Package), Octopus classifier (nl.klasse.octopus.model.IClassifier) to UML2 classifier (org.eclipse.uml2.uml.Classifier), etc. When the visitor visited one element of the input (Octopus model), it created the same element in the form of UML2 model, and stored it in corresponding map, which will be processed later by another visitor in the next visit. At the end, after the final UML2 models has been generated, it is serialized into file, using the information which stored in that map.

The following visitors are called in the given order, depending on whether the option has been set to use the given functionality. In the following we will name all visitors and what they do. The complete visitor code can be found in the appendix.

II.2.2.1 Visitor01

In the first pass, all Octopus Packages with its imported packages are read which results UML2 Packages are created accordingly. For this pass, only the package names are being read.

For each octopus classifier which can be, in general, Enumeration, Class, or Interface, results in the creating of the corresponding UML2 classifier by its name.

The creation of UML2 properties is done by visiting the octopus attributes. The octopus attribute is mapped to property in the UML2.

The same happens with the creation of UML2 operation and UML2 parameter when it visits the same elements of the Octopus model. Basically all the UML2 elements which needed, is created by its name in this first pass.

II.2.2.2 Visitor02

In the second pass, the subpackages are linked to the correct package based on its relation of the octopus model. The same thing is done for the classifier, each of them is connected with its owner. As an addition, the visitor will traverse all the primitive type, and attach it to its owner package.

The Enumeration type will complete in this stage which its literal is created based on the Octopus Enumeration model. For another type of Octopus classifier (IClassifier) which is not part of Enumeration, after checking the type of its attribute, then it will create the correct model of the UML2 classifier, and linked it back with the Octopus classifier using the maps. The generalization of each classifier is finalized here.

The other property of the UML2 Operation is determined, the return type and the parameters.

The property of Property (visibility) and Parameter (direction) is set and will be completed.

II.2.2.3 Visitor03

After having the basic model set, all association related elements are built in this third pass (last pass). The UML2 association created based on the association model of the input Octopus UML.

II.2.2.4 UML2 File Writer

Here we create a resource set with the specified URI, add the package to the resource contents, and ask the resource to save itself using the default options. The error will display in the console if an exception occurs.

II.2.2.5 Output in UML2

All of those code bundled in one plugins named octopus2uml2. To get it run in Eclipse, create a new plugins project and put the source code inside source folder of the project (Figure II-5). Then launch new workspace, and run the OUML2 -> Convert to UML2 from the context menu of the input Octopus project (Figure II-6).

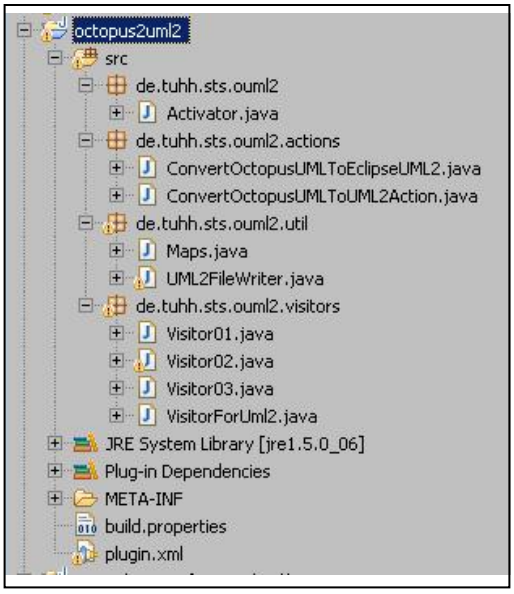

Figure II-5. Octopus2uml2 plugins

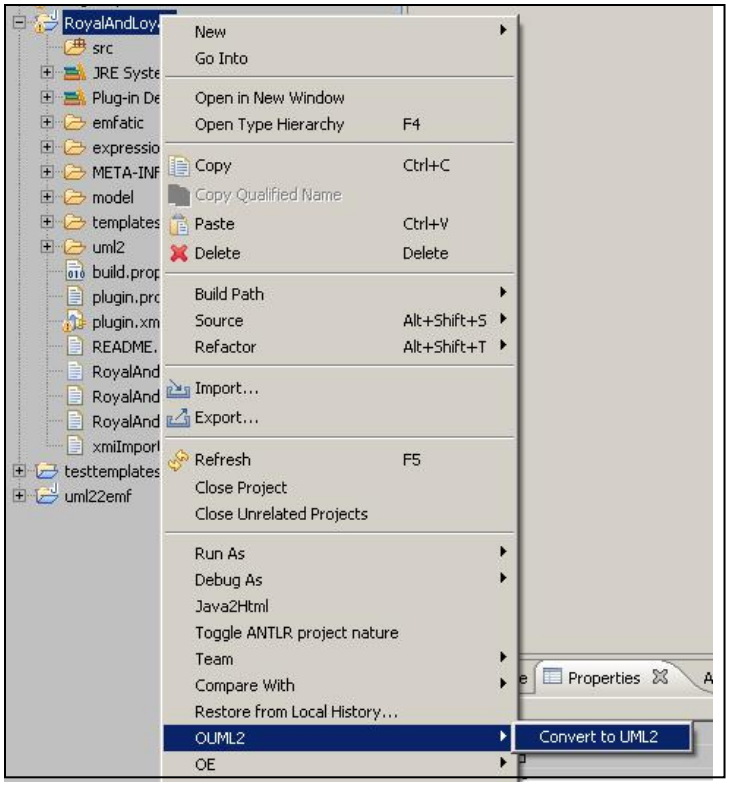

Figure II-6. Conversion Action from Octopus UML model into UML2 model

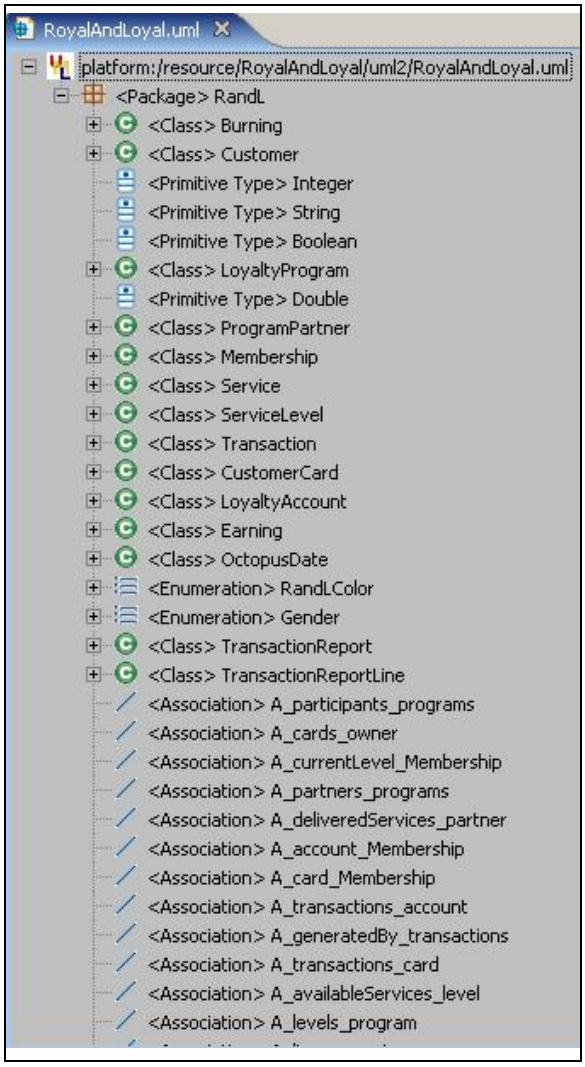

Figure II-7. UML2 model of Royal and Loyal

Figure II-7 shows the result of Royal and Loyal in UML2 model, generated from the action in Figure II-6. This result is similar with one in Figure II-4 only differs in the representation of the model.

II.2.2.6 Output in omondo

Omondo provides the visual presentation of the UML2 model. From UML2 model, we can obtained the ecore model which can be opened using Omondo (http://www.omondo.com) and resulting the visual diagram of the model. This output

(Figure II-8) is useful to check the correctness of the output temporarily by comparing it with one in Figure II-1.

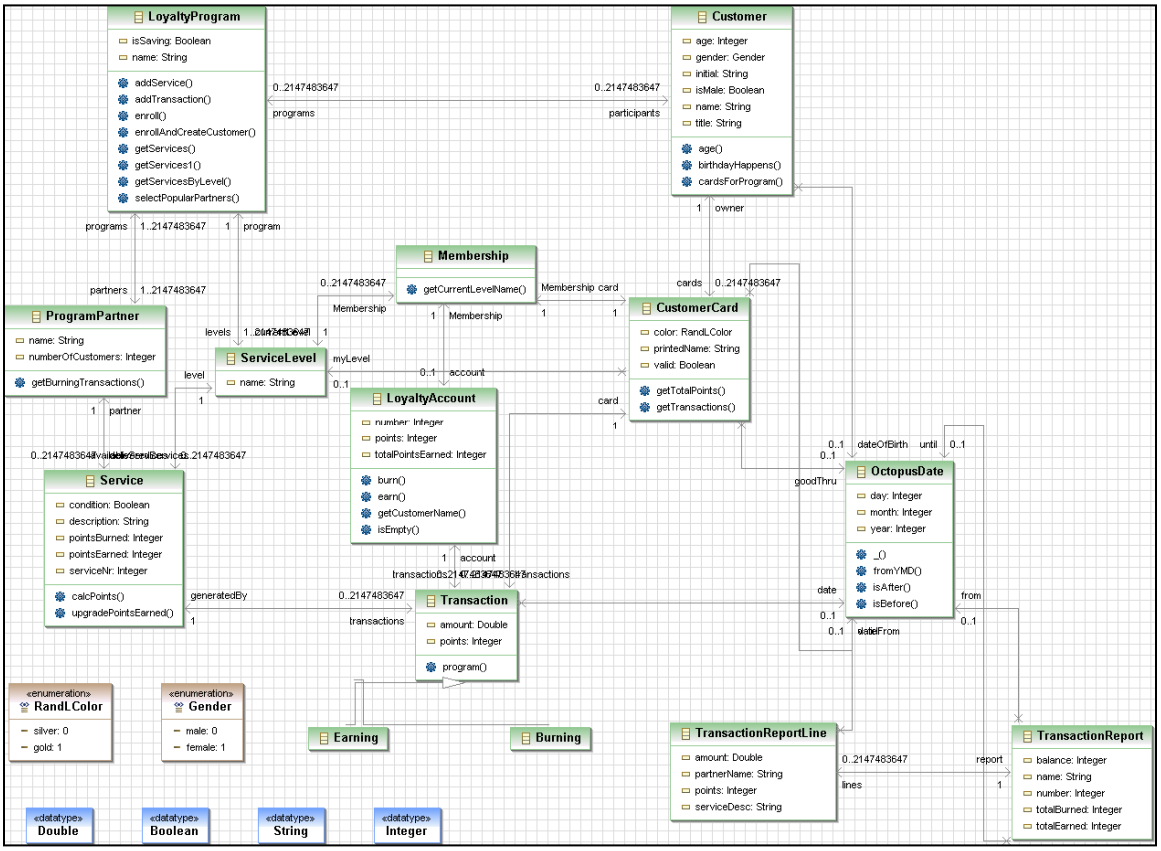

Figure II-8. Royal and Loyal in UML2 model visually built using Omondo

II.3 UML2 Codegen

Once we had the UML2 model, the next things to do is to get the generated code based on the model defined. UML2 has provided UML2 Codegen API in their frameworks.

The purpose of the UML2 code generation extensions is to enhance the default EMF code generator to handle concepts that are specific to UML, such as subsetting, redefinition, and derived unions. A generator model created from a UML model that makes use of any of these concepts (assuming the options were enabled in the wizard) will produce code that enforces the associated constraints. The customized templates do

provide a few other options (such as factory and look-up methods) that were considered too application-specific to be included in EMF, but these customizations could easily be added independently of the UML2 code generator [10].

The UML2 Codegen is not a stand alone module which can be used as generator like EMF Codegen, but furthermore it is the extension of EMF Codegen. However, we can use it as provided by the eclipse framework like the following.

Having the UML2 model, we can start the EMF project wizard by choosing it as the importer model and enabling the UML-specific options (i.e. process redefining operations/properties, subsetting properties, and/or union properties). From EMF project we can directly derive the generated code by choosing 'generate model' in the context menu of the genmodel.

Following are the steps to get the generated code out of the UML2 model.

1. Create new EMF Project with UML2 as the importer model

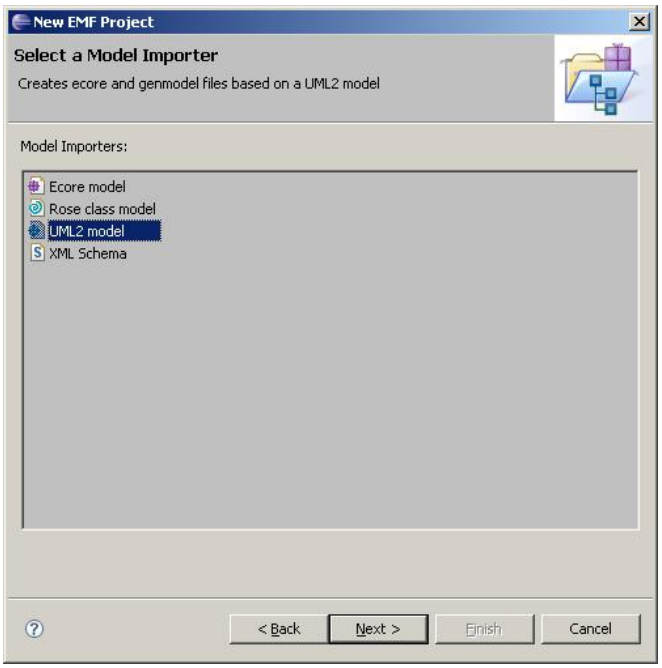

Figure II-9. UML2 model as importer model

2. Specify the model and enable the UML2 specific options, or simply press the process all button.

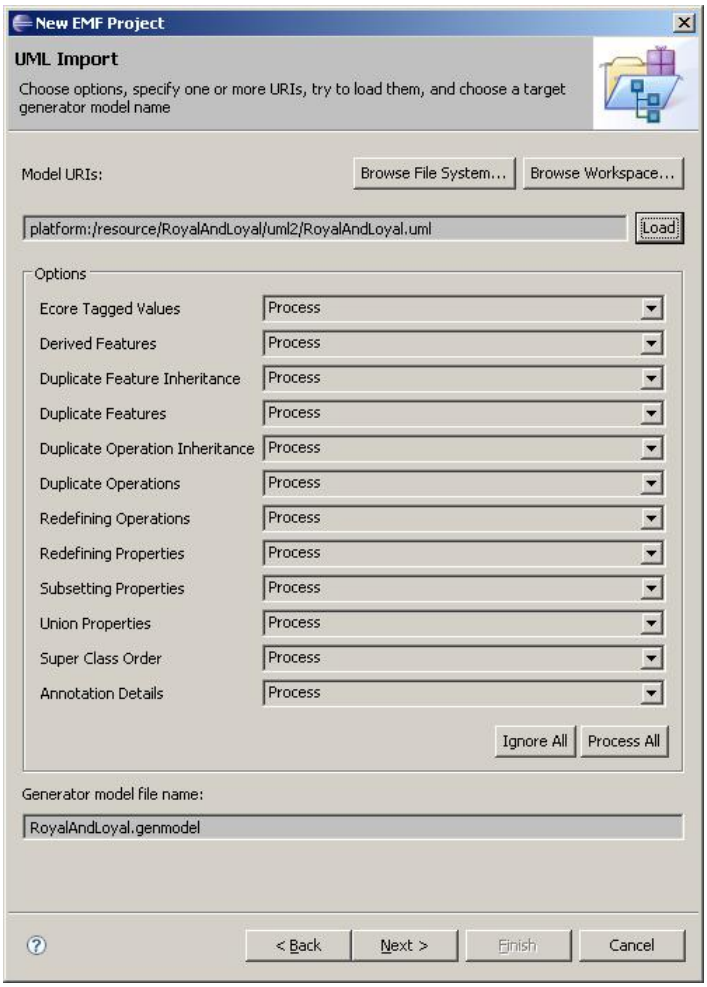

Figure II-10. Process All the options

3. After having the .genmodel file, open it and run Generate Model Code from its context menu.

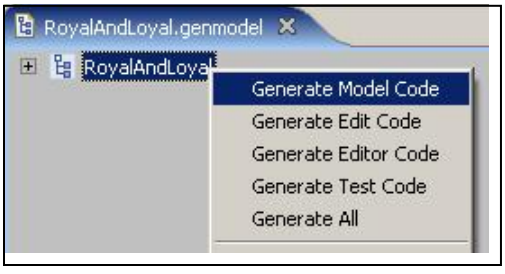

Figure II-11.Generate Model Code

4. The result code is generated in src folder

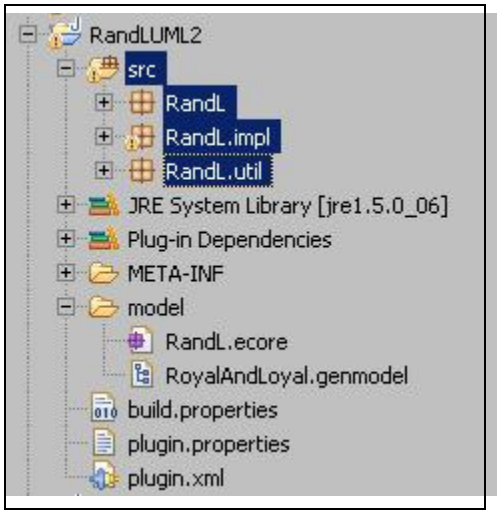

Figure II-12. Generated Code from UML2 model

II.4 Summary

This StA used one of the example Octopus projects (Royal and Loyal, which modeled a computer system for a fictional company) as a case study for the conversion to Eclipse UML2.

UML2 model contains three major categories of elements: classifier, events, and behaviors. The conversion from Octopus model to UML2 is done by creating visitor to visit all the Octopus element and create the corresponding UML2 element. Each visitor stored the information (Octopus element and UML2 element) into maps. At the end, the UML2 model, which can be obtained from that maps, is serialized into UML file.

UML2 Codegen which is based on EMF Codegen cannot be used separately without EMF. To get the generated code out of the UML2 model, use the EMF Codegen instead, with all the options enabled.

III Adding Constraints

The conversion from Octopus model to UML2 is not complete. There is still something missing in the generated UML2 model. The constraints which are represented in OCL expressions have not been transformed into UML2 model.

There are several ways to have it expressed in the resulting UML2 model. In the following we will discuss some alternatives for such translation.

III.1 Adding Constraints by Creating Constraints Element in UML2

A constraint can be specified by creating a Constraint element and specifying its constrained element(s) and specification (e.g. an opaque expression). The UML2 project does not provide any specific UI to support OCL, but it is possible to create constraints via the UML editor [9].

Following are the steps to create the constraint elements. We will continue from the model which has been generated, RandL.uml (UML2 model).

1. Open Royal and Loyal UML2 model with UML Model Editor in eclipse.
2. Create a Constraint of Customer classifier.

Figure III-1. Create a constraint element

3. Add the attribute as one of the constraint's constrained elements (from the Properties view)

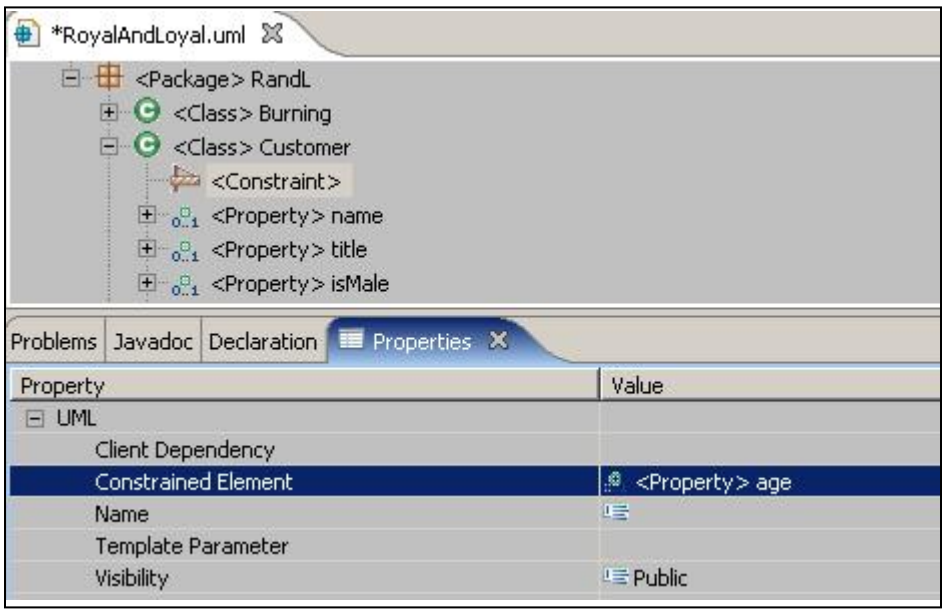

Figure III-2. Add the constrained element properties

4. Create an opaque expression as its specification

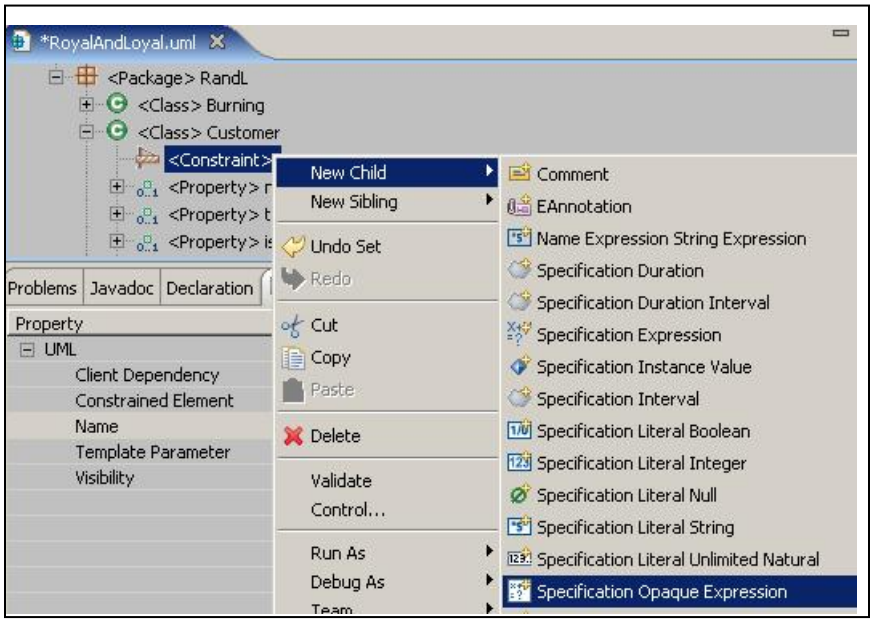

Figure III-3. Create an opaque expression to the constraint

5. Add a body string to the expression

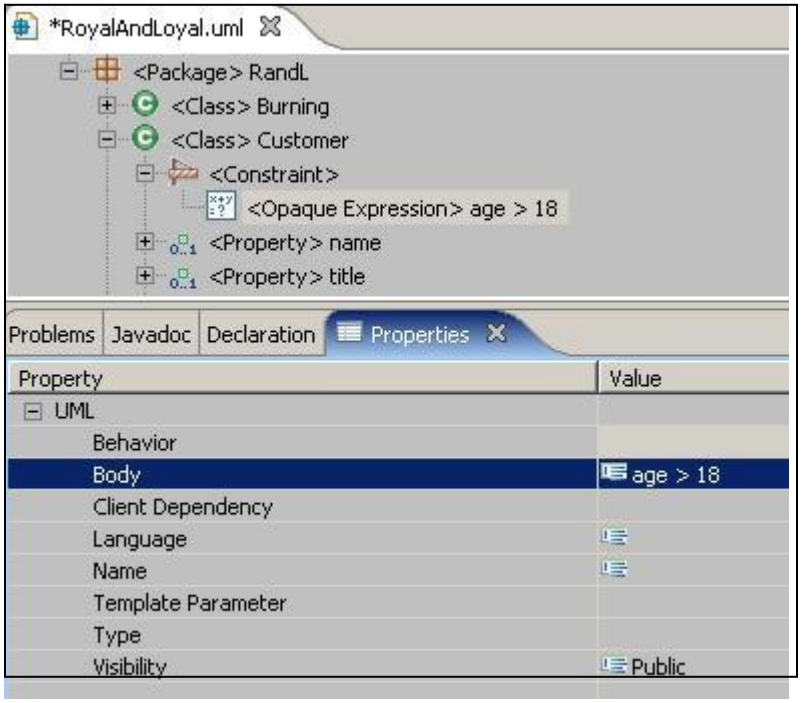

Figure III-4. Add value to the body properties of the constraint

6. Result after adding name of the Constraints and the Expression

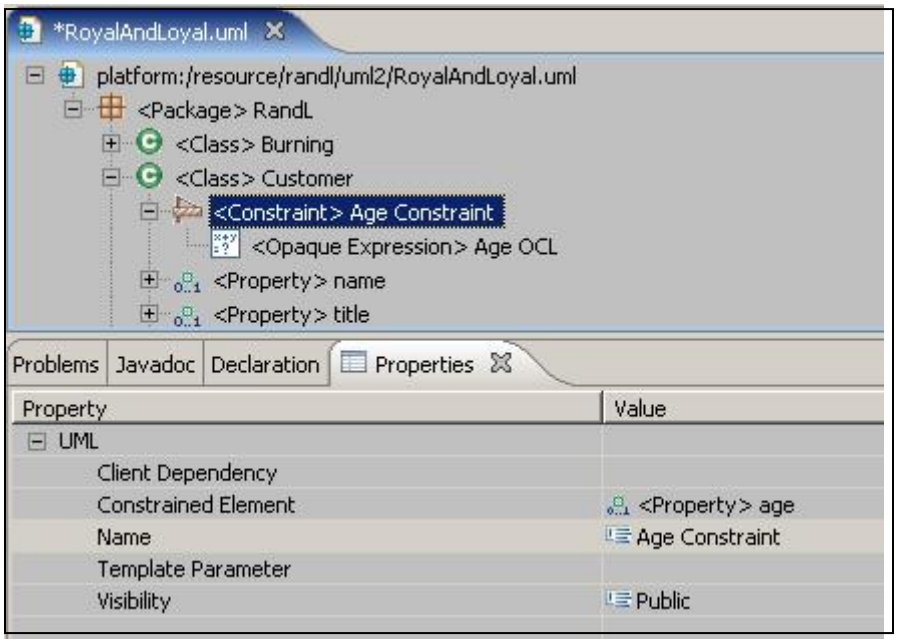

Figure III-5. The results after the constraints is added

Now the constraint element has been added in the UML2 model. We would like to see whether this constraints will be generated to the code, by converting it to Ecore model then generate the code using EMF Codegen.

7. Convert it into Ecore Model

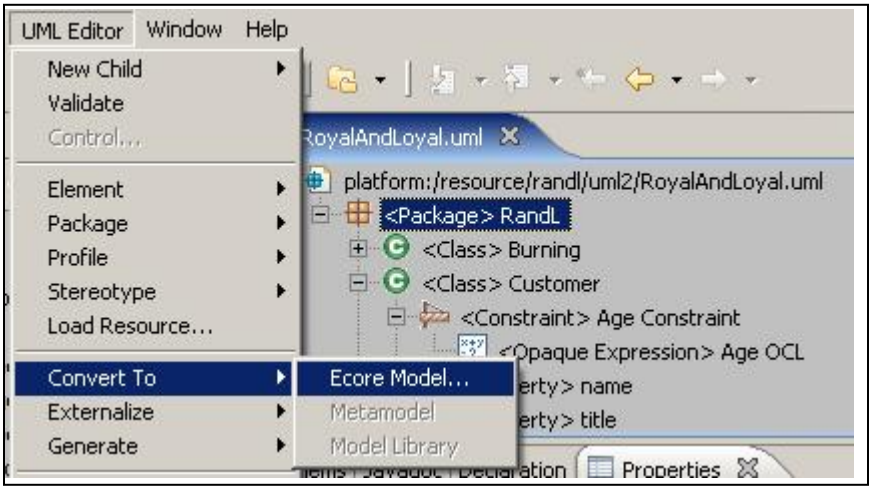

Figure III-6. Converting to Ecore Model

8. The constraints has been added as additional validation method.

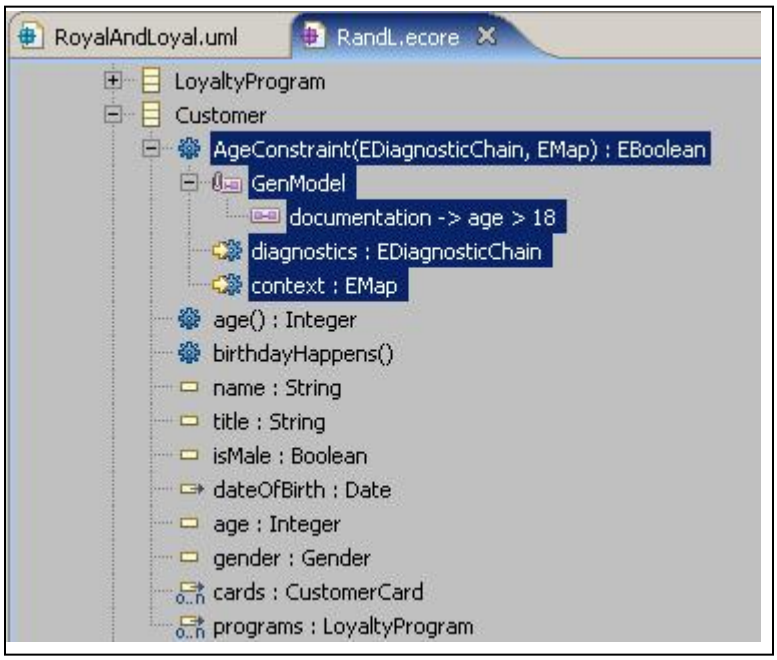

Figure III-7. The Ecore model after adding the constraint

There is also the annotation attached to the validation method. That annotation informs us that, the OCL expression will be added as the documentation of the method.

9. Here is the generated code from .genmodel file, which can be obtained from the .ecore file.

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* <!-- begin-model-doc -->
* age > 18
* <!-- end-model-doc -->
* @model
* @generated
*/
boolean AgeConstraints(DiagnosticChain diagnostics, Map context);
```

Figure III-8. Customer.java

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public boolean AgeConstraints(DiagnosticChain diagnostics, Map context) {
// TODO: implement this method
// -> specify the condition that violates the invariant
// -> verify the details of the diagnostic, including severity and message
// Ensure that you remove @generated or mark it @generated NOT
   if (false) {
      if (diagnostics != null) {
          diagnostics.add (new BasicDiagnostic (Diagnostic.ERROR,
             RandLValidator.DIAGNOSTIC_SOURCE,
             RandLValidator.CUSTOMER__AGE_CONSTRAINTS,
             EcorePlugin.INSTANCE.getString(
                "_UI_GenericInvariant_diagnostic",
                new Object[] { "AgeConstraints",
                   EObjectValidator.getObjectLabel(this, context) }),
              new Object [] { this }));
       }
      return false;
   }
   return true;
}
```

Figure III-9. CustomerImpl.java

The generated validation method still needs to be completed manually by hand. The OCL constraints that have been expressed in the previous steps only generated as the

documentation in the interface (Figure III-8). This limitation makes us to find another approach in adding the constraint to the UML2 model.

III.2 EMFT Validation

The Eclipse Modeling Framework Technology project was initiated to incubate new technologies that extend or complement EMF (http://www.eclipse.org/emft). There are several topics which are covered beyond this EMFT project, some examples include Validation, OCL, Query, Transaction, and many more. Each topic has similar intention which is to coordinate all things surround the EMF. The Validation and OCL will be discussed deeper in this work.

The EMF validation framework provides a means to evaluate and ensure the well-formedness of EMF models. Validation comes in two forms: batch and live. While batch validation allows a client to explicitly evaluate a group of EObjects and their contents, live validation can be used by clients to listen on change notifications to EObjects to immediately verify that the change does not violate the well-formedness of the model [14]. The purpose of this framework is to check the integrity of the EMF metamodels by providing a generic and extensible framework which can be used in defining constraints to be checked againts the model. Actually EMF already provided EValidator API in their package (this is briefly explained in IV.2). This framework differs from it in several important aspect [14]:

1. Support for automatic validation on transaction boundaries: constraints can indicate that they are evaluated in "live" mode, as changes are made in a model, rather than by user demand.
2. Dynamic extensibility: the framework is not based on code generation.
3. Pluggable support for constraint languages such as OCL.

   However, both of them can work simultaneously to achieve the expected results.

Using this approach, the Constraints will not be generated inside the model. We will create the validation extension that will be invoked by the generated code when the validation occur.

III.2.1 Adding Constraints

We can create an EValidator implementation that delegates to the validation framework, to provide user-demand "batch mode" validation from an EMF editor or even a “live mode” validation.

Following are steps in how to implement this way using our example Royal and Loyal model:

1. Create the EMF project based on Royal and Loyal UML2 Model, choose the UML model as the model importer and add all the options (by clicking ‘Process All’) to include all the functionality of UML2 model.
2. Generate the model code, edit code, and editor code out of it.
3. Create new plugin project to provide the validation.

Figure III-10 shows the result on the workspace after adding several code supporting the validation. These code is based on the Validation Adapter Example and OCL Validation Example. The explanation is provided in their tutorial which you can get after installing EMFT Validation plugins on Eclipse.

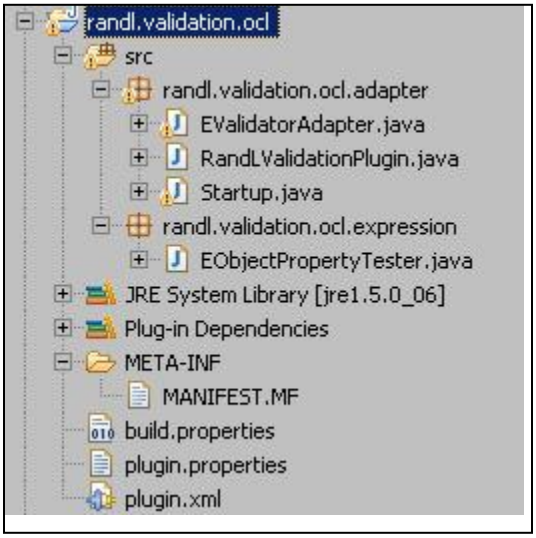

Figure III-10. Validation PluginProject

The EValidator implementation will delegate to the EMF Validation Framework to evaluate all active constraints on a sub-tree of a model. The metamodel that will be the target is the Royal and Loyal Metamodel example which are just created in previous steps.

EMF provides extension points on which to register resource factories for file extensions and EPackages for namespace URIs. However, there is no extension point on which we can register our EValidator implementation for the Royal and Loyal Metamodel. Instead, we will create an org.eclipse.ui.startup extension to register our validator when the Eclipse platform launches.

```
<plugin>
      <extension point="org.eclipse.ui.startup">
            <startup class="randl.validation.ocl.adapter.Startup"/>
      </extension>

.
.
.
</plugin>
```

Figure III-11. Startup class extension point – plugin.xml

Then the Startup class (Figure III-12) which is defined above (Figure III-11) will do the rest by registering the validator to Evalidator.Registry. With this way, each time, user invoke EMFs validate action in the context menu of Royal and Loyal Editor, this validator will be run.

```
public class Startup implements IStartup {

      /**
      * Install the validator.
      */

      public void earlyStartup() {
            EValidator.Registry.INSTANCE.put(
                  RandLPackage.eINSTANCE, new EValidatorAdapter());
      }
}
```

Figure III-12. Startup Class – Startup.java

Figure III-13 show the constraints part which has to be defined in plugin.xml. Each constraints that wants to be evaluated has to be added to this plugin.xml. This document only gives one OCL Constraints as the example which defines that the age of Customer, which is the target, has to be greater than 18. Some variables used can be defined in plugin.properties (Figure III-14).

```
<extension point="org.eclipse.emf.validation.constraintProviders">
      <category name="%category.name"
                  id="emf-validation-example/ocl">
                        %category.description
      </category>
      <constraintProvider cache="true">
         <package namespaceUri="http:///RandL.ecore"/>
         <constraints categories="emf-validation-example/ocl">
            <constraint
                  lang="OCL"
                  severity="ERROR"
                  mode="Live"
                  name="%customer1.name"
                  id="customer1"
                  statusCode="101">
               <description>%customer1.desc</description>
               <message>%customer1.msg</message>
               <!-- This constraint applies to Customer -->
               <!--<target class="Customer"/> -->
               <target class="Customer">
                  <event name="Set">
                        <feature name="age"/>
                  </event>
                     </target>
            <!-- Age of the customer should not less than 18 -->
               <![CDATA[ age >= 18 ]]>
            </constraint>
         </constraints>
      </constraintProvider>
   </extension>
```

Figure III-13. Constraint Definition – plugin.xml

```
category.name = Royal and Loyal OCL Constraints
category.description = OCL Category for Royal and Loyal

customer1.name = Customer OCL Constraint in batch mode
customer1.desc = Customer OCL Constraint description
customer1.msg = age of "{0}" has to be greater than 18
```

Figure III-14. variable definition – plugin.properties

This approach has succeeded on generation of OCL constraints into the model. In order to do that we have to put all the constraint one by one into the plugin.xml which is not a good idea for some rather large application.

III.3 EMFT OCL

The EMFT OCL component provides capabilities for queries, constraint parsing, constraint validation and content assist for user models. It defines the API for constructing, validating, and evaluating OCL queries and constraints on EMF model elements. (http://www.eclipse.org/emft/projects/ocl/)

Using this component, we can provide OCL expression as annotations element in the Ecore model. The generated model classes will have methods implementing these as described in the next few sections.

III.3.1 Adding Constraints

This technique used is based on an EMFT OCL article [2] which explains how to implement model integrity using EMFT OCL. We will follow that approach to get the OCL expression tranformed into the EMF model and evaluated at runtime.

The basic idea on adding the constraints is having them as the annotations in the Ecore model so that later on the can be transformed (by reflective inspection of that model) into code in charge of runtime checks. The mechanics of the conversion involve EMF codegen together with additional JET templates (http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html). The conversion process starts from the Octopus model and ends as the EMF model using emfatic as intermediate notation. Figure III-15 shows the chain of the conversion process: Octopus model -> Emfatic -> EMF model -> Generated Code.

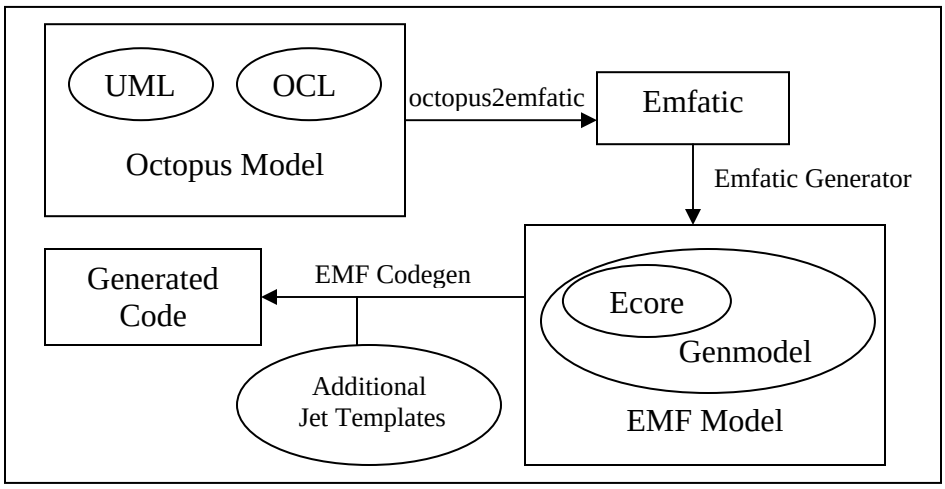

Figure III-15. Flow chain of the conversion process

Additional JET Templates are needed to generate the validation operation body. Scripting statements in these templates parse the annotations containing the constraints. At runtime, the constraint is available as a String, which is interpreted to obtain a result. The templates may also generate additional support fields.

III.3.1.1 Octopus Model to Emfatic

This process is done by extending the octopus2emfatic plugins. It already transformed the Octopus model to emfatic, only without having the constraints in it. These additional visitors will add all the constraints of the input model into the emfatic representation. Following are the four steps involved:

1. InvariantsVisitor

```
public void class_Before(IClassifier c) {
   if ( c instanceof ClassifierImpl && !(c instanceof IEnumerationType)) {
      EmfClass nc = (EmfClass) maps.classifiers.get(c);
      ClassifierImpl in = (ClassifierImpl) c;
      List<String> invNames = new ArrayList<String>();
      Iterator it = in.getInvariants().iterator();
      int invOperCount = 1;
      while(it.hasNext()) {
         IOclContext cont = (IOclContext) it.next();
         if (cont != null && (cont.getType() == OclUsageType.INV)) {
            if (cont.getName() != null && cont.getName().length() > 0) {
               if (!invNames.contains(cont.getName())) {
                  invNames.add(cont.getName());
                  addInvariant(nc, cont, invOperCount);
               }
            } else {
               addInvariant(nc, cont, invOperCount);
               invOperCount++;
            }
         }
      }
   }
}
```

Figure III-16. InvariantsVisitor.java

This visitor will visit every classifier and check whether they have OCL specification of invariants. If so, it loops over each invariant, creating an EMF annotation for each. The name of the invariant is given by its name, if any, otherwise the general ‘invariant_<classifier_name>_<order_number>’ name is used. For invariant annotation, it uses ‘invariant’ as the key and the OCL expression as the value.

2. DerivationAttributeVisitor

Very similar to the previous visitor, only this time a check is made for each Octopus attribute whether it has an OCL specification of init and derivation. If so, it will create the derivation annotation for corresponding attribute. The key of the annotation in this case is ‘derive’ and its value is the OCL expression.

```
public void attribute(IAttribute a) {
   if (a instanceof AttributeImpl) {
      AttributeImpl attr = (AttributeImpl) a;
      EmfStructuralFeature na = maps.attributes.get(a) != null ?
                                    maps.attributes.get(a) : maps.references.get(a);

      if(na != null){
         // init expression
         IOclContext cont = attr.getInit();
         if (cont != null) {
            addDerivationRule(na, cont);
         } else {
            // derivation rule
            cont = attr.getDerivationRule();
            if (cont != null) {
               addDerivationRule(na, cont);
            }
         }
      }
   }
}
```

Figure III-17. DerivationAttributeVisitor.java

3. OperationExpressionVisitor

This visitor visits all the Operations and checks whether there is any OCL specifications of pre, post, or body defined. For each type of constraint, it creates one additional operation named ‘pre_<operation_name>’, ‘body_<operation_name>’, or ‘post_<operation_name>’, depending on the specifications. For each of those additional operations, it will define one annotation which contains it as constraint. At the end, it creates one more annotation on the operation itself which contains of the invocation of those additional method defined in OCL expression.

There are two distinct types of annotations created here. First annotations use literal ‘body’ as the key, and the OCL expression as the value. This OCL expression will be checked and parsed before it is included in the body of its operation. Second annotation use ‘genmodel’ as the key, and the invocation of pre, body, or post operation, which has been previously defined, as the value. The latter type of annotation is used

when we want to put the literal value of the annotation directly into the generated body of operations without having it parsed as OCL constraint.

```
public void operation_Before(IOperation o) {
   if (o instanceof OperationImpl) {
      EmfOperation no = maps.operations.get(o);
      OperationImpl oper = (OperationImpl) o;

      Iterator preIt = oper.getPreConditions().iterator();
      int preCount=0;
      while( preIt.hasNext()) {
         IOclContext preCont = (IOclContext) preIt.next();
         if (preCont != null && !preCont.hasErrors()) {
            preCount++;
            createConditionOperation(no, preCont, "pre_", preCount);
         }
      }

      Iterator postIt = oper.getPostConditions().iterator();
      int postCount=0;
      while( postIt.hasNext()) {
         IOclContext postCont = (IOclContext) postIt.next();
         if (postCont != null && !postCont.hasErrors()) {
            postCount++;
            createConditionOperation(no, postCont, "post_", postCount);
         }
      }

      if(preCount==0 && postCount==0){
         IOclContext bodyCont = oper.getBodyExpression();
         if (bodyCont != null) {
            addBody(no, bodyCont);
         }
      }else{
         IOclContext bodyCont = oper.getBodyExpression();
         if (bodyCont != null) {
            createConditionOperation(no, bodyCont, "body_", 0);
            addConditionToBody(no, preCount, 1, postCount);
         }else{
            addConditionToBody(no, preCount, 0, postCount);
         }
      }
   }
}
```

Figure III-18. OperationExpressionVisitor.java

4. Add all those visitor to the convert method of ConvertUMLToEmfatic Class

5. Launch the octopus2emfatic plugin by running it as new eclipse application (runtime workbench), then invoke the OE -> Convert To Emfatic of the Royal and Loyal octopus project

```
public class ConvertUMLToEmfatic {

   public static StringBuffer convert(IPackage p, boolean isTop, IOclLibrary
oclLib ) {

         StringBuffer sb = new StringBuffer();
         if (p.getSubpackages().size() == 1) {
            p = (IPackage) p.getSubpackages().iterator().next();
         }

         VisitorForEmfatic v;

         v = new Visitor01(null);
         p.accept(v);
         v = new Visitor02(v.maps);
         p.accept(v);
         v = new Visitor03(v.maps);
         p.accept(v);

         v = new InvariantsVisitor(v.maps);
         p.accept(v);
         v = new DerivationAttributeVisitor(v.maps);
         p.accept(v);
         v = new OperationExpressionVisitor(v.maps);
         p.accept(v);

         EmfPackage top = v.maps.packages.get(p);
         String str = top.toString();
         sb.append(str);
         return sb;
   }
}
```

Figure III-19. ConvertUMLToEmfatic.java

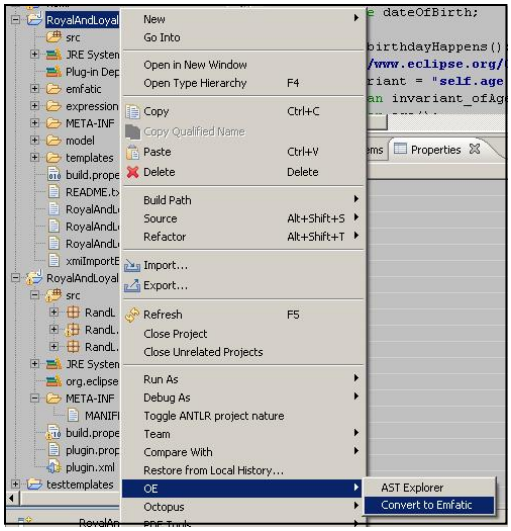

Figure III-20. Convert to Emfatic

6. This is the result of generated emfatic after extending the octopus2emfatic plugin. The invariant is listed as the validator method and notice its annotation (green marks) which represents the OCL expression.

```
class Customer {
    attr Integer age;
    attr Boolean isMale;
    @"http://www.eclipse.org/OCL/examples/ocl"(
        derive = "self.name.substring(1, 1)")
    readonly volatile transient derived attr String initial;
    attr Gender gender;
    attr String name;
    attr String title;

    val OctopusDate dateOfBirth;

    @"http://www.eclipse.org/OCL/examples/ocl"(
        invariant = "self.age >= 18")
    op boolean invariant_ofAge(ecore.EDiagnosticChain diagnostics,ecore.EMap context);

    @genmodel(
        body = "// TODO: implement this method
    // Ensure that you remove @generated or mark it @generated NOT body
    assert post_birthdayHappens_1();
    ")
    op void birthdayHappens();
    @"http://www.eclipse.org/OCL/examples/ocl"(
        body = "self.age = self.age + 1")
    op boolean post_birthdayHappens_1();

    .
    .
    .
    .

    !ordered ref CustomerCard[0..*] #owner cards; /*  end + CustomerCard.cards [0..*]
in assoc cards_owner + CustomerCard.cards [0..*] <-> + Customer.owner [1..1] */
    !ordered ref LoyaltyProgram[0..*] #participants programs; /*  end +
LoyaltyProgram.programs [0..*] in assoc Membership + Customer.participants [0..*] <->
+ LoyaltyProgram.programs [0..*] */

}
```

Figure III-21. Royal And Loyal emfatic model - RoyalAndLoyal.emf

III.3.1.2 Emfatic to Generated Code

1. Create the ecore model out of the emfatic

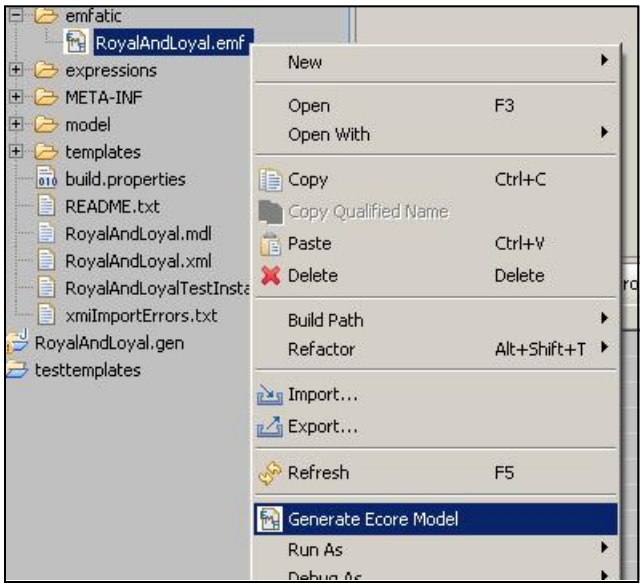

Figure III-22. Generate Ecore Model from Emfatic

2. From ecore files we can get the EMF model (.genmodel). Choose the 'EMF model', and select 'ecore' as the importer model. Then continue the instruction until we can get the .genmodel.

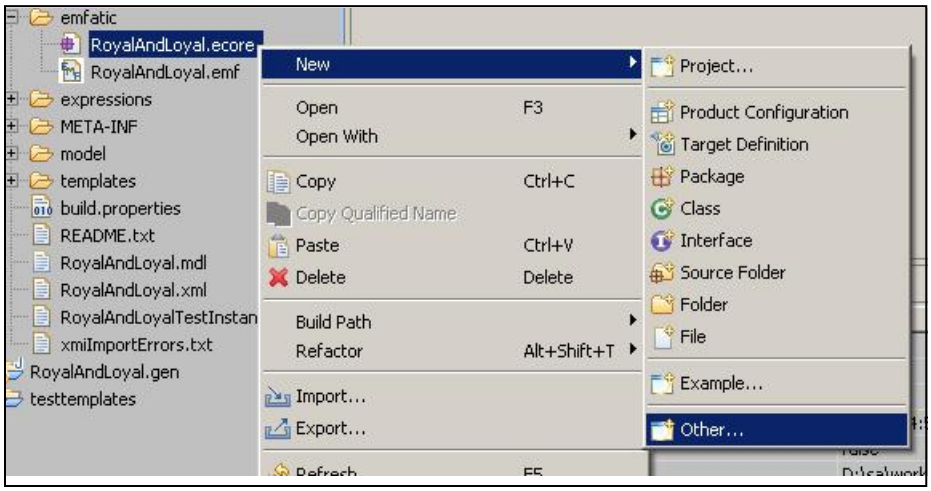

Figure III-23. Generate genmodel from Ecore model

3. Normally, from this model, we can directly obtain the generated code by invoking 'Generate Model Code' from context menu of this model. But, since we have to

implement the model integrity in this case by adding the OCL constraint into the generated code, we have to configure some JET templates of this model. The template is obtained from this resource [2]. There are few things to configure on the properties of the model. See the green marks.

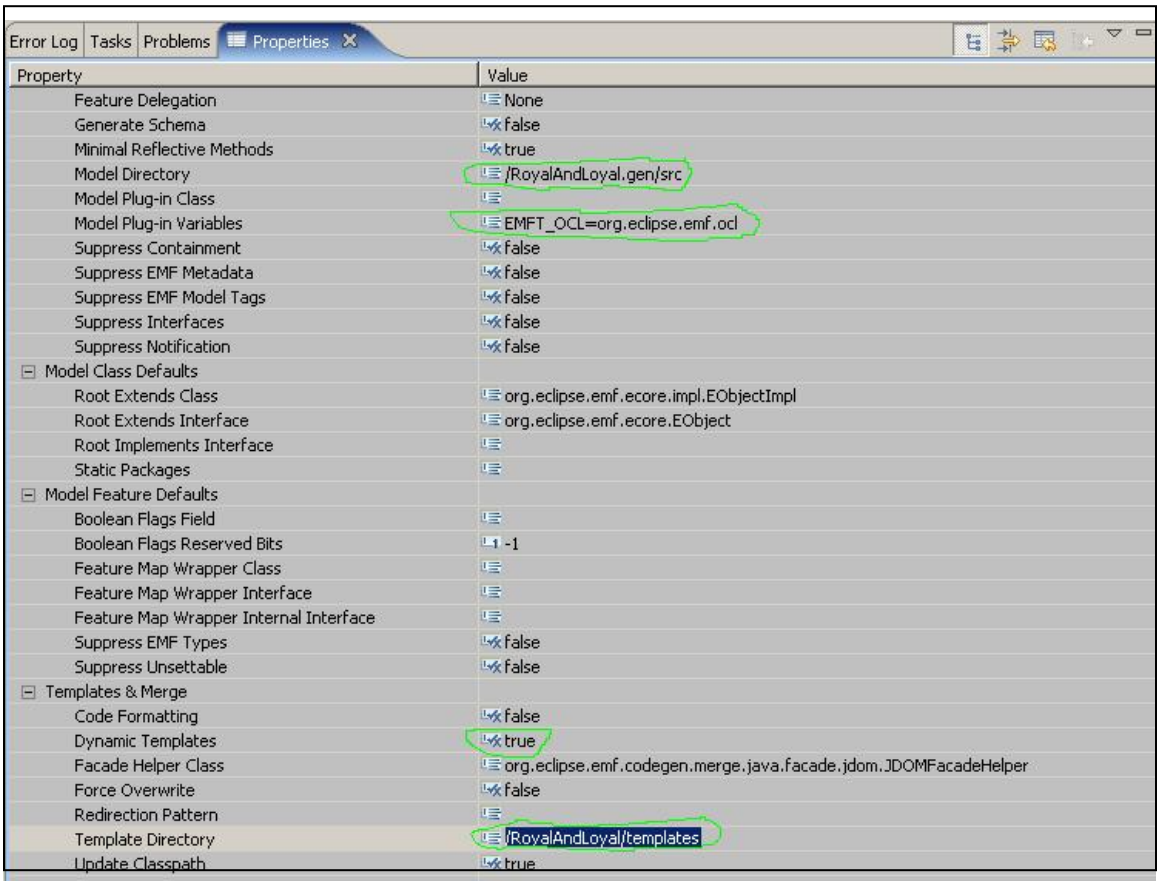

Figure III-24. Properties of RoyalAndLoyal.genmodel

4. After all things are configured, we can generate the code

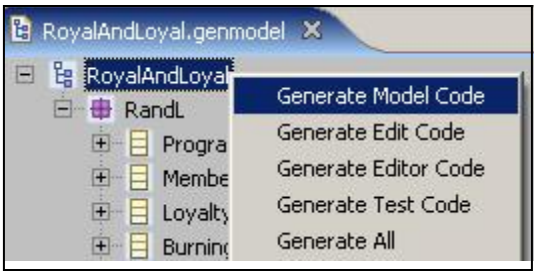

Figure III-25. Generate the Model Code

5. Here is the generated code

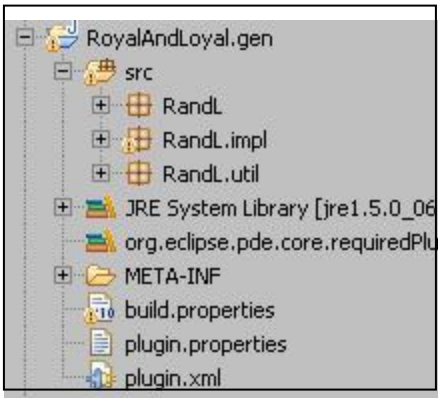

Figure III-26. Generated code on package view

Using this approach, finally we got the OCL expression as part of the body of the validator method. It means we have successfully generated the constraints which is defined in the model into the code. Figures below show the generated code of the constraints.

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public boolean pre_isEmpty_1() {
   if (pre_isEmpty_1BodyOCL == null) {
      EOperation eOperation = (EOperation)
      eClass().getEOperations().get(4);
      Environment env =
        ExpressionsUtil.createOperationContext(eClass(), eOperation);
      EAnnotation ocl =
        eOperation.getEAnnotation(OCL_ANNOTATION_SOURCE);
       String body = (String) ocl.getDetails().get("body");
       try {
          pre_isEmpty_1BodyOCL =
            ExpressionsUtil.createQuery(env, body, true);
       } catch (ParserException e) {
       throw new
            UnsupportedOperationException(e.getLocalizedMessage());
       }
   }

   Query query =
      QueryFactory.eINSTANCE.createQuery(pre_isEmpty_1BodyOCL);
   EvalEnvironment evalEnv = new EvalEnvironment();
   query.setEvaluationEnvironment(evalEnv);
   return ((Boolean) query.evaluate(this)).booleanValue();
}
```

Figure III-27. Pre Constraint generated Code –LoyaltyAccountImpl.java

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public Boolean isEmpty() {
  assert pre_isEmpty_1();
  // TODO: implement this method
  // Ensure that you remove @generated or mark it @generated NOT body
  assert post_isEmpty_1();
  throw new UnsupportedOperationException();
}
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public boolean post_isEmpty_1() {
.
.
.
}
```

Figure III-28. Post Constraint Generated Code – LoyaltyAccountImpl.java

```
/**
* The parsed OCL expression for the definition of the '{@link #invariant_ofAge <em>Invariant
of Age</em>}' invariant constraint.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @see #invariant_ofAge
* @generated
* invariant = self.age >= 18
 */
private static OCLExpression invariant_ofAgeInvOCL;
private static final String OCL_ANNOTATION_SOURCE =
"http://www.eclipse.org/OCL/RoyalAndLoyal/ocl";
public boolean invariant_ofAge(DiagnosticChain diagnostics, Map context) {
   if (invariant_ofAgeInvOCL == null) {
      EOperation eOperation = (EOperation) eClass().getEOperations().get(1);
      Environment env = ExpressionsUtil.createClassifierContext(eClass());
      EAnnotation ocl = eOperation.getEAnnotation(OCL_ANNOTATION_SOURCE);
      String body = (String) ocl.getDetails().get("invariant");
      try {
         invariant_ofAgeInvOCL = ExpressionsUtil.createInvariant(env, body, true);
      } catch (ParserException e) {
         throw new UnsupportedOperationException(e.getLocalizedMessage());
      }
   }
   Query query = QueryFactory.eINSTANCE.createQuery(invariant_ofAgeInvOCL);
   EvalEnvironment evalEnv = new EvalEnvironment();
   query.setEvaluationEnvironment(evalEnv);
   if (!query.check(this)) {
      if (diagnostics != null) {
         diagnostics.add
            (new BasicDiagnostic
               (Diagnostic.ERROR, RandLValidator.DIAGNOSTIC_SOURCE,
                   RandLValidator.CUSTOMER__INVARIANT_OF_AGE,
                    EcorePlugin.INSTANCE.getString("_UI_GenericInvariant_diagnostic",
                             new Object[] { "invariant_ofAge",
                                            EObjectValidator.getObjectLabel(this, context) }),
                    new Object [] { this }));
      }
      return false;
   }
   return true;
}
```

Figure III-29. Validator Method – CustomerImpl.java

III.4 Summary

In this stage we already had the converted UML2 model transform from Octopus Model, only without the Constraints included. There are several ways to get them expressed in the destination model. First approach is by adding it as constraint element using UML2 editor, which later can be generated automatically programmatically. With this way, we found out that the constraint is not generated as the implementation body of the validation operation, instead only stated as documentation, which needs to manually typed by hand. The second approach is using EMFT validation. With this way we create the extension of the model which based on the validation framework. All the constraints has to be put on plugin.xml of that extension package, which requires too much effort in such big scale application. The third approach involves EMFT OCL components. With this way we put the constraint as the annotation which later EMF Codegen with additional JET Templates can process them to be generated as proper code. Using this solution, we can get all the constraint expressed correctly in the code.

IV EMF.Edit & EValidator API

IV.1 EMF.edit

The purpose of EMF.edit [13] is to build very functional viewers and editors for the model. Basically, with this framework, developers which use EMF model in their application can easily generate an editor that will display and edit instances of the model using standard JFace viewers and a property sheet.

EMF.Edit is an Eclipse framework that includes generic reusable classes for building editors for EMF models. It provides:

1. Content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets.
2. A command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.
3. A code generator capable of generating everything needed to build a complete editor plug-in for your EMF model. It produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors.

   For More Information about EMF.edit please refer to [3].

IV.2 Evalidator API

EMF Codegen generates package validator classes for each package which has invariant method in its classes. EObjectValidator is the base for all generated package validator classes. This base class provides validation on such aspects [12]:

1. The actual multiplicities of the attributes and references match the bounds defined in the model.
2. The defined data type of the attributes is respected.
3. Any cross referenced objects are contained in resources.
4. Every proxy is properly resolved.

IV.3 Generation of an Editor for Royal and Loyal

After having the model ready in EMF, now we can generate the editor to manipulate the instance. This generation editor can also be used to ensure the integrity of the model by validate action. This validate action will invoke the validate method provided by package validator classes (EValidator API).

1. Generate edit code and editor code of Royal and Loyal

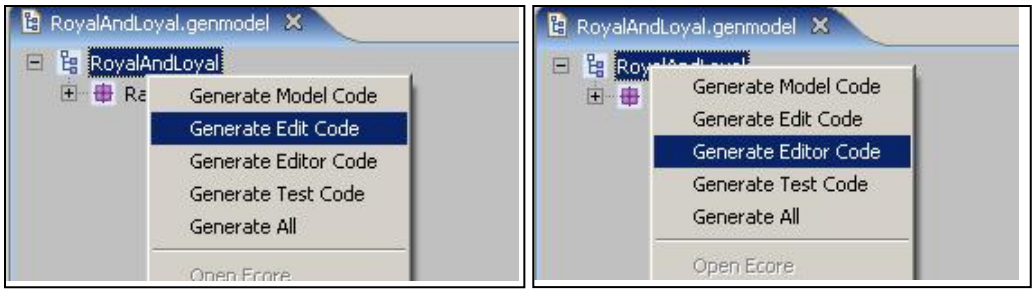

Figure IV-1. Generate Edit and Editor Code

2. Here is the result on the workspace. There are 3 projects created based on EMF Codegen. The .edit and .editor package is used to provide the diagram editor and manipulation of the instance of the model.

Figure IV-2. Generated Code of EMF Codegen

3. Launch a new workspace to instantiate the model. For detail, see step 4 of [5]

4. In this editor we can create an instance of the model. Then manipulate the properties in the properties view (provided in the context menu). Figure IV-3 shows the editor of Royal and Loyal.

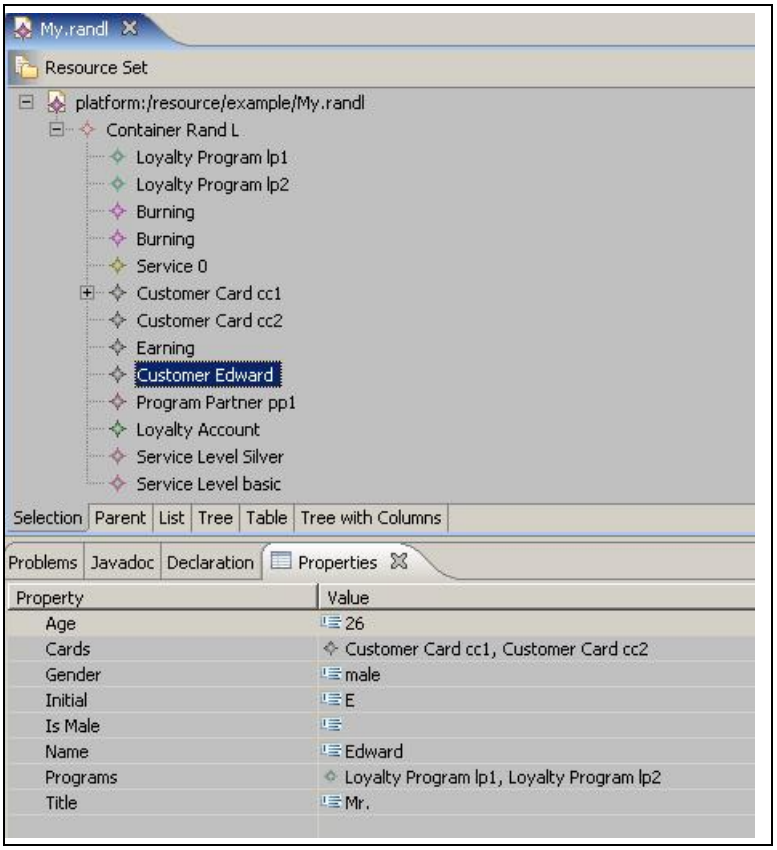

Figure IV-3. Editor of Royal and Loyal

5. In this editor we can validate the instance of the model

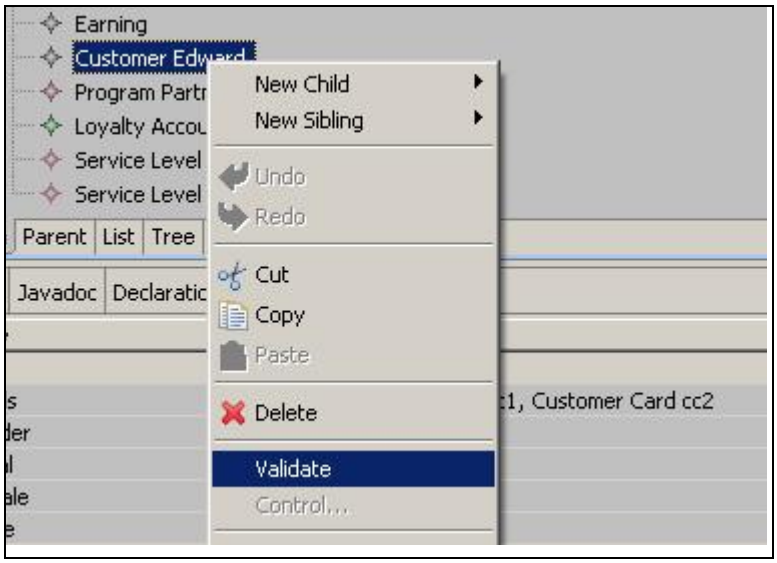

Figure IV-4. Validate Action - Context Menu of Editor

This action will invoke the validate method of the package validator classes which is generated by EMF codegen.

```
/**
* Calls <code>validateXXX</code> for the corresonding classifier of the model.
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/

protected boolean validate(int classifierID, Object value, DiagnosticChain
diagnostics, Map context) {

   switch (classifierID) {
   .
   .
   .
   case RandLPackage.CUSTOMER:
        return validateCustomer((Customer)value, diagnostics, context);
   .
   .
   .
   default:
      return true;
   }
}
```

Figure IV-5. validate method – RandLValidator.java

Figure IV-5 is generated for RandL package. It is implementing the validate method of its base class EObjectValidator.

```
/**
* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public boolean validateCustomer(Customer customer, DiagnosticChain diagnostics, Map
context) {
boolean result = validate_EveryMultiplicityConforms(customer, diagnostics, context);
if (result || diagnostics != null)
   result &= validate_EveryDataValueConforms(customer, diagnostics, context);
if (result || diagnostics != null)
   result &= validate_EveryReferenceIsContained(customer, diagnostics, context);
if (result || diagnostics != null)
   result &= validate_EveryProxyResolves(customer, diagnostics, context);
if (result || diagnostics != null)
   result &= validateCustomer_invariant_Customer1(customer, diagnostics, context);
if (result || diagnostics != null)
   result &= validateCustomer_invariant_Customer2(customer, diagnostics, context);
.
.
.
return result;
}
```

Figure IV-6. validateCustomer method – RandL.java

This method will invoke each method which states in the validateXXX in the Figure IV-6 which belongs to Evalidator API. This invocation will preserve all of the integrity of the specific instance.

IV.4 Runtime Aspect

Once we had the EMF model, we can create the editor to manipulate or validate the instance. After validating the instance against all the constraint defined in the model, we can see how many OCL expressions can still be preserved by the EMF model, comparing to the original model which was in form of Octopus model. This result can also be one of the parameters which can tell us whether this solutions can be improved in the future.

One thing that needs to be noticed, the EMF model does not support the Association Class. If there is any Association Class from the Octopus model, it will be treated as the normal class. In our example, Royal and Loyal, there is one association class named Membership. This affects the number of constraints that can be processed by the EMF model

This subchapter will show the results of how many OCL expressions can be processed by the Royal and Loyal EMF model. The specification of OCL expression which can be checked in this test only limited on: ‘inv’, ‘def’, ‘init’, and ‘derive’. The other three specifications (‘pre’, ‘body’, and ‘post’) can not be included in the test because they require method invocation while currently the editor can not provide it.

We will show the results based on several criteria. The X-axis denotes the context of the OCL and the Y-axis denotes specification of the OCL. (b=before, a=after)

1. All OCL expressions are included.

In this criteria, we include all the constraint in the test. In the editor, we create the instances as the requirements of the model, and validate theses instances to see whether all the constraints can be preserved. From Figure IV-7, the total percentage of the OCL expression that the EMF model can process is only 45.99 %. That is such a poor result. If we drill down, we will see the constraint that involves the random OCL expressions,

which is not relevant with this model, makes the most contribution to the failed process. The next result will remove them out. The example of this Constraints is, "inv: Set { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } } ->isEmpty()". The Association class also will be removed in the next test, because EMF model does not recognize it.

	Inv		def		init		derive		Total		
	b	a	b	a	b	a	b	a	b	a	(%)
Burning	6	6							6	6	100.00
Customer	16	11	4	1					20	12	60.00
CustomerCard	7	3	1	1	3	3	2	2	13	9	69.23
LoyaltyAccount	3	3			3	3	2	2	8	8	100.00
LoyaltyProgram	27	7	3	2					30	9	30.00
Membership	9	0	1	0					10	0	0.00
ProgramPartner	6	6	1	1					7	7	100.00
Service	7	7							8	8	100.00
ServiceLevel	19	0							19	0	0.00
Transaction	4	4							4	4	100.00
TransactionReport	2	0					5	0	7	0	0.00
TransactionReportLine							5	0	5	0	0.00
	106	47	10	5	6	6	14	4	137	63	45.99

Figure IV-7. Test Results – All OCL Expressions are included

2. Only Relevant OCL Expressions

	inv		def		init		derive		Total		
	b	a	b	a	b	a	b	a	b	a	(%)
Burning	6	6							6	6	100.00
Customer	12	11	3	1					15	12	80.00
CustomerCard	7	2	1	1	3	3	1	0	12	6	50.00
LoyaltyAccount	3	3			3	3	2	2	8	8	100.00
LoyaltyProgram	9	7	2	2					11	9	81.82
ProgramPartner	6	6	1	1					7	7	100.00
Service	7	7							8	8	100.00
ServiceLevel	1	0							1	0	0.00
Transaction	4	4							4	4	100.00
TransactionReport	2	0					3	0	5	0	0.00
TransactionReportLine							5	0	5	0	0.00
	106	47	10	5	6	6	14	4	82	60	73.17

Figure IV-8. Test Results – Only The relevant OCL expressions included

Now we can see the total percentage is much better. Analyzing this result, we can divide the cause of the failed validation into 2 categories:

1. Derived value from the unspecified parent

It is failed because the parent value has not been initialized yet, but it is already evaluated by the editor. For example, printedName, one of the field of the CustomerCard, is derived from concatenation of title and name of its owner which is Customer. In the editor, when the CustomerCard is instantiated, the corresponding field is automatically evaluated without having specified who is the 'owner' of the CustomerCard which causes null pointer exception.

This kind of error can be avoided by creating some additional condition to the model. It will not be occurred if the deriving process done when the parent class has been initialized.

2. Invocation of empty method

The constraint invokes one of the method of its context which is still empty. Normally, the empty body operation by default is filled with throwing an Exception by EMF Codegen.

Similar to first case, it can be avoided if the completion of operation body is done before the validation occur. In this case, we just let the the implementation operation body empty.

IV.5 Summary

EMF.edit generates an editor in which we can manipulate an instance of the model as well and validate that instance against the specified OCL constraints. This feature can be seen as a rapid prototyping mechanism to easily test the integrity of a model instance. The EValidator API is the underlying API internally invoked to carry out this task. Package validation classes are generated automatically to invoke the validate operation of its base class, if any.

Royal and Loyal EMF model did validate in the first attempt all the constraints but the coverage of them was not good, because most of the failed constraints were not relevant to the target model or were not supported by the target OCL interpreter

(remember the reduction operation on a collection of literals). After the problematic constraints were removed, results around coverage of constraints improved significantly. As we saw, some of the constraints that still could not be processed had to do with non-initialized data that the constraint depends on.

V Outlook

V.1 Summary

The main motivation behind this student project was to complete a tool chain from Octopus Model to UML2 model, where each tool has its own advantages. This tool chain provides the flexibility to gain the benefits provided by both of tools, Octopus and EMF. EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model, while UML2 is an EMF-based implementation of the UML 2.0 metamodel for the Eclipse platform.

EMF supports regenerating the model after changing the code but still preserves the changes of the customized code. EMF also provides the framework which gives the possibility to create, edit, and delete the instance of the model using the visual editor. This framework, known as EMF.edit, additionally can be used as validation tool to ensure the integrity of the model. Those are specific improvements that we can get with EMF model but not with Octopus model.

Once we have created the EMF model, we can have them worked with the other EMF-based components. This interoperability with other components is also one of the main advantages after converting the model into the EMF technology space. With this feature, the enhancement for the model can easily be obtained.

This student project has succeeded in converting the Octopus model into UML2 model without the constraints included (because of lack of framework support for this at time from part of Eclipse UML2). And this work also completes the convertion of Octopus model to EMF model *including* the OCL constraints. The integrity of the generated model that can be preserved is at least more than 70% which was tested using the generated editor provided by EMF.

V.2 Future Work

There are so many works to complete, generally in MDA world, and in this project in particular. Currently, EMF model does not support the Association class. There

must be a way to solve this problem, for example by refactoring the original model into one amenable to direct translation to EMF. This refactoring should also involve the OCL constraints that refer to the original association class.

In the algorithm to convert the octopus model, octopus2emfatic, there are some bugs remain. This code will fail to process when the input model has the same classifier's name with the emfatic's keyword, such as 'Date'. The other problem, that has been found, will occur when the model has method overloading, two or more methods have the same name with different signature. The Ecore generator, which generates emfatic to ecore file, does not support it. That would be a good progress if those bugs can be fixed in the future.

The most important enhancement is to complete the tool that can process this modeling chain smoothly and can be customized easily by the other developers. It will give a meaningful contribution to the eclipse community which already has big support from and for the developer.

Appendixes

VI Visitor Code of Octopus2uml2

VI.1 de.tuhh.sts.ouml2.visitors package

VI.1.1 VisitorForUml2.java

```
package de.tuhh.sts.ouml2.visitors;

import nl.klasse.octopus.model.IClassifier;
import nl.klasse.octopus.model.IStructuralFeature;
import nl.klasse.octopus.model.VisibilityKind;
import nl.klasse.octopus.modelVisitors.DefaultPackageVisitor;
import nl.klasse.octopus.stdlib.internal.types.StdlibCollectionType;
import org.eclipse.uml2.uml.MultiplicityElement;
import de.tuhh.sts.ouml2.util.Maps;

public class VisitorForUml2 extends DefaultPackageVisitor {

      protected String baseUri =
                  "http://de.tuhh.sts.octopus/octopus2emfatic/2006";

      public Maps maps = new Maps();

      public VisitorForUml2(Maps m) {
            if (m != null) {
                  this.maps = m;
            }
      }

      protected void setModifiersFor(MultiplicityElement
                        multiplicityElement, IStructuralFeature osf) {

            if (osf.getType() instanceof StdlibCollectionType) {
                  StdlibCollectionType ct = (StdlibCollectionType)
                                                      osf.getType();
                  multiplicityElement.setIsOrdered(ct.isOrderedSet() ||
                                                ct.isSequence());
                  multiplicityElement.setIsUnique(ct.isSet() ||
                                                ct.isOrderedSet());
            }
      }

      protected void setCardinalityFor(MultiplicityElement
                        multiplicityElement, IStructuralFeature osf) {

            multiplicityElement.setLower(
                              osf.getMultiplicity().getLower());
            int up = osf.getMultiplicity().getUpper();
            if (up == Integer.MAX_VALUE) {
```

```
                up = -1;
            }
            multiplicityElement.setUpper(up);
        }

        protected String getStrTypeExprFor(IClassifier theType) {
            String str = theType.getName();
            if (str.trim().toLowerCase().equals("real")) {
                str = "Double";
            }
            return str;
        }

        protected org.eclipse.uml2.uml.VisibilityKind getVisibility
                                (VisibilityKind octopusVisibilityKind){

            if(octopusVisibilityKind == VisibilityKind.NONE)
                return
                org.eclipse.uml2.uml.VisibilityKind.PACKAGE_LITERAL;
            else if(octopusVisibilityKind == VisibilityKind.PRIVATE)
                return
                org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERAL;
            else if(octopusVisibilityKind == VisibilityKind.PROTECTED)
                return
                org.eclipse.uml2.uml.VisibilityKind.PROTECTED_LITERAL;
            else if(octopusVisibilityKind == VisibilityKind.PUBLIC)
                return
                org.eclipse.uml2.uml.VisibilityKind.PUBLIC_LITERAL;
            else
                return
                org.eclipse.uml2.uml.VisibilityKind.PACKAGE_LITERAL;
        }

}
```

VI.1.2 Visitor01.java

```
package de.tuhh.sts.ouml2.visitors;

import java.util.Iterator;

import nl.klasse.octopus.model.IAttribute;
import nl.klasse.octopus.model.IClassifier;
import nl.klasse.octopus.model.IInterface;
import nl.klasse.octopus.model.IOperation;
import nl.klasse.octopus.model.IPackage;
import nl.klasse.octopus.model.IParameter;
import nl.klasse.octopus.model.internal.types.EnumerationTypeImpl;
import nl.klasse.octopus.model.internal.types.ImportedElementImlp;
import nl.klasse.octopus.model.internal.types.OperationImpl;
import nl.klasse.octopus.oclengine.IOclContext;
import nl.klasse.octopus.stdlib.internal.types.StdlibCollectionType;
import org.eclipse.emf.common.util.EMap;
import org.eclipse.emf.ecore.EAnnotation;
import org.eclipse.uml2.uml.Classifier;
import org.eclipse.uml2.uml.Operation;
```

```
import org.eclipse.uml2.uml.Package;
import org.eclipse.uml2.uml.Parameter;
import org.eclipse.uml2.uml.Property;
import org.eclipse.uml2.uml.UMLFactory;
import de.tuhh.sts.ouml2.util.Maps;

public class Visitor01 extends VisitorForUml2 {

  public Visitor01(Maps m) {
    super(m);
  }

  @Override
  public void package_Before(IPackage octopusPackage) {

    Package uml2Package = UMLFactory.eINSTANCE.createPackage();
    uml2Package.setName(octopusPackage.getName());
    maps.packages.put(octopusPackage, uml2Package);

    for (Object io : octopusPackage.getImports()) {
      ImportedElementImlp importedElement = (ImportedElementImlp) io;
      org.eclipse.uml2.uml.Package importedPackage =
                              UMLFactory.eINSTANCE.createPackage();
      importedPackage.setName(importedElement.getPathname().toString());
    }

    super.package_Before(octopusPackage);
  }

  @Override
  public void class_Before(IClassifier octopusClassifier) {

    Classifier uml2Classifier;

    if (octopusClassifier instanceof EnumerationTypeImpl) {
      uml2Classifier = UMLFactory.eINSTANCE.createEnumeration();
    } else {
      uml2Classifier = UMLFactory.eINSTANCE.createClass();
      uml2Classifier.setIsAbstract(octopusClassifier.getIsAbstract());
    }

    maps.classifiers.put(octopusClassifier, uml2Classifier);
    uml2Classifier.setName(octopusClassifier.getName());

    super.class_Before(octopusClassifier);
  }

  @Override
  public void interface_Before(IInterface octopusInterface) {

    Classifier uml2Classifier = UMLFactory.eINSTANCE.createInterface();
    uml2Classifier.setName(octopusInterface.getName());
    maps.classifiers.put(octopusInterface, uml2Classifier);
    super.interface_Before(octopusInterface);
  }

  @Override
```

```
public void attribute(IAttribute octopusAttribute) {

  Property uml2Property = UMLFactory.eINSTANCE.createProperty();
  maps.properties.put(octopusAttribute, uml2Property);
  uml2Property.setName(octopusAttribute.getName());

  // modifiers and cardinality

  setModifiersFor(uml2Property, octopusAttribute);
  setCardinalityFor(uml2Property, octopusAttribute);
  super.attribute(octopusAttribute);
}

@Override
public void operation_Before(IOperation octopusOperation) {

  Operation uml2Operation = UMLFactory.eINSTANCE.createOperation();
  uml2Operation.setName(octopusOperation.getName());

  maps.operations.put(octopusOperation, uml2Operation);

  // modifiers and cardinality

  if (octopusOperation.getReturnType() instanceof
                                        StdlibCollectionType) {
    StdlibCollectionType ct = (StdlibCollectionType)
                                  octopusOperation.getReturnType();
    uml2Operation.setIsOrdered(ct.isOrderedSet() || ct.isSequence());
    uml2Operation.setIsUnique(ct.isSet() || ct.isOrderedSet());
    uml2Operation.setUpper(-1);
  }

  if (octopusOperation.isOclDef()) {
    addOCLExpression(octopusOperation, uml2Operation);
  }

  super.operation_Before(octopusOperation);
}

@Override
public void parameter(IParameter octopusParameter) {

  Parameter uml2Parameter = UMLFactory.eINSTANCE.createParameter();
  uml2Parameter.setName(octopusParameter.getName());

  // type will be placed in Visitor02

  if (octopusParameter.getType() instanceof StdlibCollectionType) {
    StdlibCollectionType ct = (StdlibCollectionType)
                                    octopusParameter.getType();
    uml2Parameter.setIsOrdered(ct.isOrderedSet() || ct.isSequence());
    uml2Parameter.setIsUnique(ct.isSet() || ct.isOrderedSet());
    uml2Parameter.setUpper(-1);
  }

  maps.parameters.put(octopusParameter, uml2Parameter);
  super.parameter(octopusParameter);
```

```
  }
}
```

VI.1.3 Visitor02.java

```
package de.tuhh.sts.ouml2.visitors;

import java.util.List;

import nl.klasse.octopus.model.IAttribute;
import nl.klasse.octopus.model.IClassifier;
import nl.klasse.octopus.model.IOperation;
import nl.klasse.octopus.model.IPackage;
import nl.klasse.octopus.model.IParameter;
import nl.klasse.octopus.model.IPrimitiveType;
import nl.klasse.octopus.model.internal.types.AttributeImpl;
import nl.klasse.octopus.model.internal.types.EnumLiteralImpl;
import nl.klasse.octopus.model.internal.types.EnumerationTypeImpl;
import nl.klasse.octopus.model.internal.types.ParameterImpl;

import org.eclipse.uml2.uml.Class;
import org.eclipse.uml2.uml.Classifier;
import org.eclipse.uml2.uml.Enumeration;
import org.eclipse.uml2.uml.Operation;
import org.eclipse.uml2.uml.Package;
import org.eclipse.uml2.uml.Parameter;
import org.eclipse.uml2.uml.ParameterDirectionKind;
import org.eclipse.uml2.uml.Property;
import org.eclipse.uml2.uml.Type;

import de.tuhh.sts.ouml2.util.Maps;

public class Visitor02 extends VisitorForUml2 {

  public Visitor02(Maps m) {
    super(m);
  }

  @Override
  public void package_Before(IPackage octopusPackage) {

    Package uml2Package = maps.packages.get(octopusPackage);

    for (Object subpo : octopusPackage.getSubpackages()) {
      IPackage octopusSubPackage = (IPackage) subpo;
      Package uml2SubPackage = maps.packages.get(octopusSubPackage);
      uml2Package.createNestedPackage(uml2SubPackage.getName());
      maps.packages.put(octopusSubPackage,
            uml2Package.getNestedPackage(uml2SubPackage.getName()));
    }

    for (Object co : octopusPackage.getClassifiers()) {
      IClassifier octopusClassifier = (IClassifier) co;
      Classifier uml2Classifier =
```

```
                    maps.classifiers.get(octopusClassifier);
    if (octopusClassifier instanceof EnumerationTypeImpl) {
      uml2Package.createOwnedEnumeration(uml2Classifier.getName());
      maps.classifiers.put(octopusClassifier, (Classifier)
                uml2Package.getOwnedMember(uml2Classifier.getName()));
    } else {
        uml2Package.createOwnedClass(uml2Classifier.getName(),
                                     uml2Classifier.isAbstract());

        maps.classifiers.put(octopusClassifier,
                    (Classifier) uml2Package.getOwnedMember(
                                     uml2Classifier.getName()));

      for(Object oo : octopusClassifier.getOperations()){
        IOperation octopusOperation = (IOperation) oo;
        IClassifier operationReturnType =
                                  octopusOperation.getReturnType();
        if(operationReturnType != null){
          if (operationReturnType instanceof IPrimitiveType) {
            if(!maps.types.containsKey(operationReturnType)){
              String str = getStrTypeExprFor(operationReturnType);
              Type uml2Type =
                      uml2Package.createOwnedPrimitiveType(str);
              maps.types.put(operationReturnType, uml2Type);
            }
          }
        }
        for(Object op : octopusOperation.getParameters()){
          IParameter octopusParameter = (IParameter) op;
          IClassifier parameterType = octopusParameter.getType();
          if (parameterType instanceof IPrimitiveType) {
            if(!maps.types.containsKey(parameterType)){
              String str = getStrTypeExprFor(parameterType);
              Type uml2Type =
                      uml2Package.createOwnedPrimitiveType(str);
              maps.types.put(parameterType, uml2Type);
            }
          }
        }
      }

      for(Object oa : octopusClassifier.getAttributes()){
        AttributeImpl octopusAttribute = (AttributeImpl) oa;
        IClassifier octopusType = octopusAttribute.getType();

        if (octopusType instanceof IPrimitiveType) {
          if(!maps.types.containsKey(octopusType)){
            String str = getStrTypeExprFor(octopusType);
            Type uml2Type = uml2Package.createOwnedPrimitiveType(str);
            maps.types.put(octopusType, uml2Type);
          }
        }
      }
    }
  }

  super.package_Before(octopusPackage);
```

```
}

@Override
public void class_Before(IClassifier octopusClassifier) {

  // handle enum and return
  if (octopusClassifier instanceof EnumerationTypeImpl) {
    Enumeration uml2Enumeration =
          (Enumeration) maps.classifiers.get(octopusClassifier);
    EnumerationTypeImpl e = (EnumerationTypeImpl) octopusClassifier;
    for (Object lo : e.getLiterals()) {
      EnumLiteralImpl l = (EnumLiteralImpl) lo;
      uml2Enumeration.createOwnedLiteral(l.getName());
      uml2Enumeration.getOwnedLiteral(l.getName()).setVisibility(
                                  getVisibility(l.getVisibility()));
    }
    uml2Enumeration.setVisibility(getVisibility(e.getVisibility()));
    return;
  }

  // from now on it's either a class or interface
  Class uml2Class = (Class) maps.classifiers.get(octopusClassifier);
  uml2Class.setVisibility(getVisibility(
                                  octopusClassifier.getVisibility()));
  for (Object ao : octopusClassifier.getAttributes()) {
    AttributeImpl octopusAttribute = (AttributeImpl) ao;

    Property uml2Property = maps.properties.get(octopusAttribute);

    Type uml2Type;
    if (octopusAttribute.getType() instanceof IPrimitiveType) {
      uml2Type = maps.types.get(octopusAttribute.getType());
    } else {
      uml2Type = maps.classifiers.get(octopusAttribute.getType());
    }

    if(uml2Type!=null){
      uml2Property.setType(uml2Type);
      uml2Class.createOwnedAttribute(uml2Property.getName(),
          uml2Type, octopusAttribute.getMultiplicity().getLower(),
          octopusAttribute.getMultiplicity().getUpper());
      maps.properties.put(octopusAttribute,
          uml2Class.getOwnedAttribute(uml2Property.getName(),
                                                        uml2Type));
    }
  }
  // extends
  List supers = octopusClassifier.getGeneralizations();
  supers.addAll(octopusClassifier.getInterfaces());
  for (Object octoSuper : supers) {
    Classifier uml2Classifier = maps.classifiers.get(octoSuper);
    uml2Class.createGeneralization(uml2Classifier);
  }

  super.class_Before(octopusClassifier);
}
```

```
@Override
public void operation_Before(IOperation octopusOperation) {

  Class uml2Class = (Class) maps.classifiers.get(
                                  octopusOperation.getOwner());
  Operation uml2Operation = maps.operations.get(octopusOperation);

  //  return type
  if (octopusOperation.getReturnType() != null) {
    Type uml2Type;
    if (octopusOperation.getReturnType() instanceof IPrimitiveType) {
      uml2Type = maps.types.get(octopusOperation.getReturnType());

    } else {
      uml2Type = maps.classifiers.get(
                             octopusOperation.getReturnType());
    }
    uml2Operation.createReturnResult("return" ,uml2Type);
    uml2Operation.setType(uml2Type);
  }

  uml2Class.createOwnedOperation(uml2Operation.getName(),
                             null, null, uml2Operation.getType());
  maps.operations.put(octopusOperation,
               uml2Class.getOwnedOperation(uml2Operation.getName(),
                                                   null, null));

  Operation operation = maps.operations.get(octopusOperation);
  operation.setVisibility(getVisibility(
                                  octopusOperation.getVisibility()));

  for (Object po : octopusOperation.getParameters()) {
    ParameterImpl p = (ParameterImpl) po;
    Parameter uml2Parameter = maps.parameters.get(p);
  }

  super.operation_Before(octopusOperation);
}

@Override
public void attribute(IAttribute octopusAttribute) {

  Property uml2Property = maps.properties.get(octopusAttribute);
  uml2Property.setVisibility(getVisibility(
                             octopusAttribute.getVisibility()));

  super.attribute(octopusAttribute);
}

@Override
public void parameter(IParameter octopusParameter) {

  Parameter uml2Parameter = maps.parameters.get(octopusParameter);

  Type uml2Type;
  if (octopusParameter.getType() instanceof IPrimitiveType) {
    uml2Type = maps.types.get(octopusParameter.getType());
```

```
    } else {
      uml2Type = maps.classifiers.get(octopusParameter.getType());
    }
    uml2Parameter.setType(uml2Type);

    uml2Parameter.setDirection(ParameterDirectionKind.get(
                        octopusParameter.getDirection().getName()));

    super.parameter(octopusParameter);
  }
}
```

VI.1.4 Visitor03.java

```
package de.tuhh.sts.ouml2.visitors;

import nl.klasse.octopus.model.IAssociation;
import nl.klasse.octopus.model.IAssociationEnd;
import nl.klasse.octopus.model.IClassifier;
import nl.klasse.octopus.model.internal.types.AssociationClassImpl;
import org.eclipse.uml2.uml.AggregationKind;
import org.eclipse.uml2.uml.Classifier;
import de.tuhh.sts.ouml2.util.Maps;

public class Visitor03 extends VisitorForUml2 {

  public Visitor03(Maps m) {
    super(m);
  }

  @Override
  public void association(IAssociation a) {
    if (a instanceof AssociationClassImpl) {
      // TODO
    }
    IAssociationEnd end1 = (IAssociationEnd) a.getEnd1();
    IAssociationEnd end2 = (IAssociationEnd) a.getEnd2();
    if(end1.getOwner() != null && end2.getOwner() != null){
      addIfNavigable(end1, end2);
    }

    super.association(a);
  }

  private void addIfNavigable(IAssociationEnd end1,
                                    IAssociationEnd end2) {

    IClassifier clas1 = end1.getOwner();
    Classifier uml2Classifier1 = maps.classifiers.get(clas1);
    IClassifier clas2 = end2.getOwner();
    Classifier uml2Classifier2 = maps.classifiers.get(clas2);

    uml2Classifier1.createAssociation(end1.isNavigable(),
        (end1.isComposite() ? AggregationKind.COMPOSITE_LITERAL :
```

```
                                    AggregationKind.NONE_LITERAL),
            end1.getName(), end1.getMultiplicity().getLower(),
                end1.getMultiplicity().getUpper(),

        uml2Classifier2, end2.isNavigable(),
            (end2.isComposite() ? AggregationKind.COMPOSITE_LITERAL :
                                    AggregationKind.NONE_LITERAL),
            end2.getName(), end2.getMultiplicity().getLower(),
                end2.getMultiplicity().getUpper());

    }
}
```

Bibliography

[1] Adrian Powell , “Model with the Eclipse Modeling Framework, Part 1”, http://www-128.ibm.com/developerworks/opensource/library/os-ecemf1/, Apr 15, 2004.

[2] Christian W. Damus, “*Implementing Model Integrity in EMF with EMFT OCL*”, http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html, IBM Rational (Canada), August 1, 2006

[3] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose, “*Eclipse Modeling Framework: A Developer's Guide*”, Addison Wesley, August 11, 2003

[4] Chris Daly, “Emfatic Language for EMF Development”, http://www.alphaworks.ibm.com/tech/emfatic, Nov 9, 2004.

[5] “Generating an EMF Model”, http://www.eclipse.org/emf/docs.php?doc=tutorials/clibmod/clibmod.html, May 31, 2006

[6] Jos Warmer and Anneke Kleppe, “Introduction to OCL”, http://www.klasse.nl/ocl/ocl-introduction.html, December 20, 2005.

[7] Jos Warmer and Anneke Kleppe, “Octopus: OCL Tool for Precise Uml Specifications”, http://www.klasse.nl/octopus/index.html, March 15, 2006.

[8] Jos Warmer, Anneke Kleppe, “*The Object Constraint Language - Getting Your Models Ready for MDA*”, Second Edition, Addison-Wesley Edition 2003.

[9] Kenn Hussey, “*How to Express OCL constraints?*”, UML2 Newsgroup, June 07, 2006.

[10] Kenn Hussey, “*UML2 Codegen*”, UML2 Newsgroup, May 23, 2006.

[11] “*The Eclipse Modeling Framework (EMF) Overview*”, http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.html, June 16, 2005.

[12] "*The Eclipse Modeling Framework (EMF) Validation Framework Overview*", http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.Validation.html, June 23, 2005.

[13] "*The EMF.Edit Framework Overview*", http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.Edit.html, June 1, 2004.

[14] "*Tutorial: EMF Validation General*", http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.gmf.doc/tutorials/msl/validationTutorial.html, June 07, 2005.

[15] "*Unified Modeling Language: Infrastructure*", version 2.0, Object Management Group, March, 2006.

[16] "*Unified Modeling Language: Superstructure*", version 2.0, Object Management Group, August 2005.

[17] "What is UML2?", http://www.eclipse.org/uml2/.