Compilation of OCL into Java for the Eclipse OCL Implementation

Master Thesis

submitted by:

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

supervised by:

Prof. Dr. Ralf Möller
Software Technology and System (STS) - TUHH

Prof. Dr. Dieter Gollmann
Sicherheit in Verteilten Anwendungen (SVA) - TUHH

M.Sc. Miguel Garcia
Software Technology and System (STS) - TUHH

Hamburg, Germany
May 2007

Abstract

The separation of Platform Independent Model (PIM) and Platform Specific Model (PSM) defined by Model Driven Architecture (MDA) results in flexibility representation in process design. UML which becomes the standard in modeling representing the PIM can fits into several implementation platforms of the systems. Afterwards PSM can be derived from PIM by either automatically performed by compiler or manually created. OCL constraints come into play to ensure, among other things, the well-formedness of the models that conform to the languages in which PIMs and PSMs are expressed

EMF (Eclipse Modeling Framework) provides facility to generate code from structured data model. Beyond that, MDT (Modeling Development Tools) which provide new technologies within modeling project in eclipse platform, extends EMF feature by providing OCL implementation. It does not compile the OCL into Java code, instead it only interprets the OCL at runtime. Due to the fact that currently there are no tools of OCL compilation involves in EMF, the OCL compiler will be a significant improvement for the EMF community. This thesis offers an implementation of compilation OCL into Java within EMF. Compiler technology approach is used in designing this OCL compiler.

Declaration

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

Hamburg, 2 May 2007
Ayatullah Jibran Shidqie

Acknowledgements

I would like to thank Prof. Dr. Ralf Möller for giving me an opportunity to do my Master Thesis at Software, Technology & Systems (STS) Department of Hamburg University of Technology, Germany.

I would also like to thank M.Sc. Miguel Garcia for providing valuable advices and guidances on this project. Without it this project might never have been realized.

Finally, I would like to thank my parents, my family, and my friends for their continuous support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

MDA (Model Driven Architecture) [Obj03] is one of the important aspects in developing an enterprise level software system. It provides a better approach to software architects in designing very well high-quality results. Evidently, the most faced challenges, i.e. reusability or unplanned system evolution, are part of the problem that could easily be solved by using MDA. Recently many industries have begun to initiate their methodology using this evolving technology. The key success factor of this architecture is modeling, which in turn makes UML (Unified Modeling Language) [Obj07] comes into play as one of the powerful modeling tools for MDA.

At a glance, a UML model diagram captures an ideal representation of the design models. In reality, more often than not, the thorough specification is ambiguously represented by UML diagram. There are expressions that can not be captured accurately by the diagram, which in some cases can cause flaws in the model. Therefore we need additional constraint to complete the model. OCL (Object Constraint Language) [Obj06b] perfectly fits into this condition. It is designed as a formal language to express side-effect-free constraints precisely within UML model. Thus, the combination of these tools makes the MDA techniques even closer to reality.

The real destination does not stop here. Combination of those tools indeed make the design process is all set. Nevertheless, the target of all of this is to have the model which is represented in PIM (Platform Independent Model) and PSM (Platform Specific Model) transformed into the executable code which shape the real system. The problem that we face today is lack of tools that could perform compilation those models together with constraints into code especially a specific, yet powerful framework, like EMF (Eclipse Modeling Framework).

This automation process, generating model into code, has been targeted by the industry for many years. The extensive effort can be focused on the high-quality design process which clearly results in a very good model design without worrying about the implementation detail of the specific language. This mechanism hides many of the complexities which possibly come and provides good abstraction to the system.

1.1.1 Why EMF?

Eclipse Modeling Framework [BSM+03] is built with the spirit of enriching the tools which support MDA. It has a very good architecture containing the structured elements which keep the developer focus on development of the model rather than implementation details. Besides, it supports not only one class of models, which includes XML schema, UML class diagram, or annotated Java interfaces, but the generated

code is also equipped with notification, referential integrity, and customizable persistence to XMI features.

The Eclipse Modeling Framework can be seen as a product of MDA development which provides a modeling and code generation framework based on structured model. In fact this framework is hybrid from Essential MOF which part of the OMG standard for metamodeling. The whole orchestration of EMF is facilitated by the core EMF framework which includes a meta model, a so called Ecore Metamodel. It describes models which can produce a set of Java classes and provides runtime support for the models including change notification and persistence support with default XMI serialization.

Currently, many successful projects already take EMF into account and they are continuing to grow considering EMF has a very prospective future.

1.2 Objective

In spite of numerous strengths that it has, EMF model by its own is not powerful enough to express the complete model behavior. It still requires constraint expressions to completely describe the precise model behavior. The problem which has not been solved is that the constraint expressions can not be elaborated with the generated code to preserve the integrity of the model. As of now, there is no capability of EMF to generate OCL expression into Java code within their framework. To fulfill the missing feature that EMF does not have at current time, this project proposes a solution to provide the compilation mechanism from OCL expression into Java expression within EMF.

Compiling OCL expression will improve EMF generated code in a way that one can get not only good quality code but also integrity of the specified model at the same time. At the end, the combination of those powerful features makes significant contribution to the growing MDA tools.

1.3 Related Work

There are several groups working on this kind of compilation. Octopus [Jos06] has provided very good solution around OCL needs. It is able to check the correctness of OCL expression syntax at compile time. And it also could perform transformation from the model with OCL into Java code. Team from TU Dresden [Tea07a] has produced OCL Compiler which has the possibility to generate Java code and SQL from the constraints written in OCL 2.0 Another team from Hungary [LLC05] has implemented the OCL compiler for .NET. However, none of them support the implementation of code generation for EMF.

EMF is supported by various Eclipse project for the needs of new technologies that extend or complement EMF along their evolving way. One which surely related to this project is called Modeling Development Tools (MDT) - OCL [Tea07b]. It is part of the MDT project of the Eclipse framework which is developed to facilitate Eclipse Modeling project in general. MDT OCL provides the implementation of the OCL OMG standard for EMF based models. In fact, almost all the requirement of OCL specification has been covered by MDT OCL. But, once again, they do not provide the facility to compile OCL expressions into Java code within EMF. Nevertheless it gives very significant help in developing this OCL compiler project.

1.4 Structure of the Work

In the next chapter we will review some prior knowledge to the core of this project. They include information about constraints and some frameworks which has already

involved with constraint expressions i.e. EMF and Octopus. Chapter 3 discusses the analysis and design process to realize the project. It introduces the approach and the architecture used in the implementation phase. Chapter 4 covers the implementation of OCL compiler from OCL expression to Java expression. At the end, closing chapter presents the summary and discusses the potential work which may come in the future.

Chapter 2

Preliminaries

2.1 Needs of constraints

UML has defined a variety of diagrams that can be chosen whenever it fits the particular requirement. All the condition must be represented in the way of the diagram can handle, most of the cases are with line, arrows, number and sometimes with supplementary text. Without realizing it, some required information could be loss in representing such model. The nature characteristic of diagrams is readable yet informal, imprecise and incomplete.

Those limitations of diagram denoting the model cause ambiguousness to the represented model. For example (this is taken from [WK03]), in the UML model as shown in figure 2.1, an association between class Flight and class Person, indicating that a certain group of persons are the passengers on a flight. It is represented in the diagram with multiplicity relationship (0..*) from class Flight to class Person, which means that the number of passengers vary from 0 to unlimited, regardless the airplane.

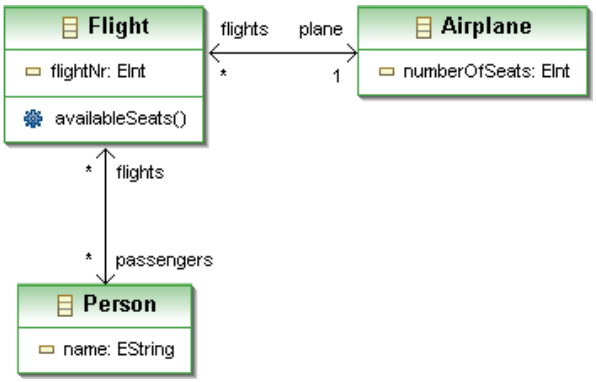

Figure 2.1: Example of model diagram which cannot express all the requirements

In the real world, the specified condition is not sufficient. The number of passengers has to be restricted to the number of seat capacities associated to the flight, otherwise it might raise an accident. This kind of condition can be patched with complementary constraints. The correct and accurate way to specify such condition is by additionally creating constraint expression, which can be represented in Object Constraint Language (OCL), to the model. The following OCL expression is an example expression to complete the specification above.

```
context Flight
inv:  passengers->size() <= plane.numberOfSeats
```

OCL offers precise and unambiguous interpretation when it is combined with UML diagrams. The example restricts the number of passenger of the flight which must be less or equal than the number of seat capacities of the corresponding plane. Without further expression, UML itself can not represent complete requirements.

2.2 OCL

This sub section introduces the summary of OCL, a formal language used to describe expressions on UML models, which is excerpted from various source, mainly from OCL specification [Obj06b] and OCL Book [WK03].

2.2.1 OCL and the Characteristics

OCL is a language which has the capability to express additional information about the model. Sometimes it is required to include the OCL expression into the model considering the limitation of UML model as previously explained. It is a compromised language which exists in between traditional formal language and natural language. Combining both of the benefits, yield a language that can express constraints precisely yet easily understandable.

The evaluation of the OCL expression could not change the state of the model. This behavior is implied by the fact that OCL is a declarative language which can simply state what should be done but not how. As a consequence, one could express what is aimed without restricting the implementation of the model.

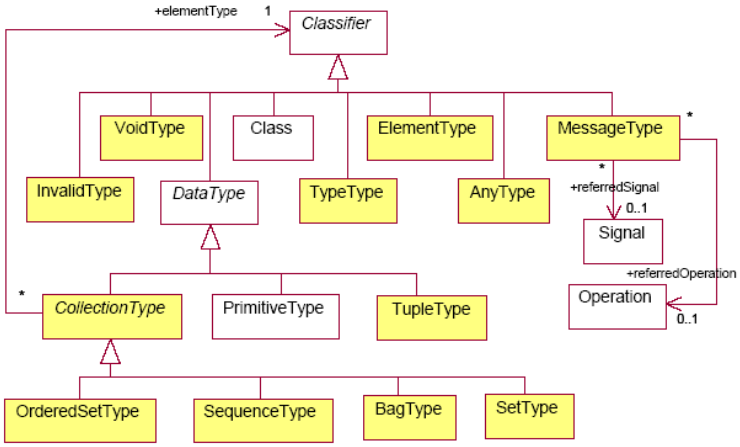

Figure 2.2: Hierarchy of OCL types

One of the essential characteristics of OCL is that it is a typed language. It means that for each OCL expression has a type. Therefore, there is a conformance rule for OCL expressions to be followed, for instance, Integer expression type can not be compared with String expression type. These rules made OCL expressions can be checked of its well formed by the specification without being executable. Figure 2.2 [Obj06b] shows the hierarchy of OCL types model.

Mostly, OCL expression is used to specify query or invariant of the model. There are more functions which can be identified by OCL expression. Each specified function

typically depends on its context. Following subsection describes about the context and the function of OCL expression in more detail.

2.2.2 Context of an OCL Expression

Each OCL expression is defined based on an element of the UML model. The element of the model entity, on which the OCL expression is defined, is identified as the context of the expression. Typically, it can be a class, interface, datatype, or operation.

The example of defining the OCL expression which has to contain context, is shown in following invariant expression:

```
context Customer
inv ofAge:  self.age >= 18
```

The keyword 'self' is used to refer explicitly to the contextual instance. The using of this keyword is optional unless when the expression is intentionally using it. In the latter case, there is no choice except to explicitly state the 'self' keyword. Example below demonstrates such condition:

```
context Membership
inv:  participants.cards.Membership.includes(self)
```

In that example, the expression requires to refer to the context as an argument to the includes() operation. Therefore, the 'self' reference must be referred within the expression.

2.2.3 Type of Function Expressions

OCL expressions consist of several function expressions. Different function expressions are defined based on its context. In this case, the suitable contexts are type of Classes and Interface, Attribute and Association End, and Operations.

Classes and Interfaces

There are 2 types of function expressions which can be defined using Class or Interface as the context: Invariants and Def expressions.

- Invariants

  An invariant is a Boolean expression that states a condition that must always be met by all instances of the type for which it is defined [WK03]. Note that, during the execution of operations, the condition does not require to be true.

  Invariant expression is indicated by the keyword 'inv' and followed by name optionally, as shown in following example:

  ```
  context Customer
  Inv myInvariant23:  self.name = 'Edward'
  ```

- Def Expressions

  OCL expression can be used to define attributes or operations of the corresponding element. As a result, every instance of the context must hold the attributes or operations defined in that expression.

  The type of the expression must conform to the type of the defined attribute or operations. Def expression is indicated by the keyword 'def'. Then it must

be followed by the name and type of the attribute or operation separated by colon. The end part of the expression is the expression itself which represent the value of the defined attribute or operation. Following are the example for Def Attribute and Def Operation respectively:

```
context Customer
def:  initial :  String = name.substring(1,1)

context CustomerCard
def:  getTotalPoints(d: Date):  Integer =
           transactions -> select (date.isAfter(d)).points->sum()
```

Note that in Def Operation expression, method name is followed by braces and it might have parameters. In that case, the type of parameters must also be included.

Attributes and Association Ends

There are 2 types of function expressions which can be defined using attribute or association end as the context: Initial values and Derivation rules.

- Initial Values

  This expression defines the initial value of the specified element. The keyword 'init' is used to indicate the expression defines the initial value expression. Example of Initial Values expression is below:

  ```
  context CustomerCard::valid
  init:  true
  ```

- Derivation Rules

  This expression defines the derivation value of the specified element. And the keyword 'derive' is used as indication of Derivation expression, like is shown in the example below:

  ```
  context CustomerCard::printedName
  derive:  owner.title.concat(' ').concat(owner.name)
  ```

  When the context is an attribute, both expressions must conform to the type of the attribute. In the case the context is an association end, when the multiplicity is at most one, it must conform to the classifier at that end, otherwise it must conform to collection type i.e. Set or OrderedSet.

  The difference between initial value and derived value lies on when they must hold the stated expression. Derived value forms an invariant. It must always have the same value with the one that the rule expresses at any point in time. Initial value has less restriction, it must hold only when the initialization of the corresponding element is occurred, in this case, when the instance is created.

Operations

There are 3 types of function expression which can be defined using Operation as the context: Precondition, Body, and Postcondition. Preconditions and postconditions share common similarities. Both of expressions are a type of Boolean expressions. And the last type, Body expression, is a type of query operation.

- Preconditions

  This expression defines the condition which must be true at the moment when the corresponding operation starts the execution. The keyword 'pre' is used to indicate Preconditions expressions. The example is shown below:

  ```
  context LoyaltyAccount::isEmpty() :  Boolean
  pre:  true
  ```

- Body

  This expression specifies the result of the corresponding operation in a single expression. Therefore, the type of the expression must conform to the type of the operation.

  Keyword 'body' is used to indicate the body expression, which can be seen in the following example.

  ```
  context LoyaltyProgram::getService() :  Set(Service)
  body:  partners.deliveredService->asset()
  ```

- Postconditions

  This expression defines the condition which must true at the moment when the corresponding operation ends the execution. The keyword 'post' is used to indicate Postconditions. The example is shown below:

  ```
  context LoyaltyAccount::isEmpty() :  Boolean
  post:  result = (points = 0)
  ```

  Note that identifier 'result' in postconditions does not represent the properties of the context. It is a special keyword of postconditions which indicates the return value from the operation.

Even though Preconditions and Postconditions share the common similarities, they are 2 different independent expressions. Of the same context, the modeler may specify multiple preconditions without specifying any postconditions and vice versa. Furthermore, when there is Preconditions and Postconditions corresponded to the same context, those are not related with each other.

2.3 Eclipse Projects

Nowadays, Eclipse [SDF+04] is not merely providing Integrated Development Environments (IDEs) functionality but also aiming at supporting the development with means of MDA. The development of several projects covering areas of Model Driven Engineering is the convincing facts that Eclipse has a strong intention in realizing their mission.

One of the top-level projects which is related to this development is called Eclipse Modeling Project (EMP). This project is concentrated on the evolution and promotion of model-based development technologies within the Eclipse community. The purpose of this project is clear, to combine the specific modeling projects under one umbrella which further can be easier to integrate or collaborate. Two interesting projects which relevant with this Thesis Project include Eclipse Modeling Framework (EMF) and Modeling Development Tools (MDT). In fact, MDT is another parent project which has several sub projects. One of the sub-projects of MDT which is important

to be acquainted with is OCL which later in this paper will be named Eclipse OCL to avoid confusion with the real OCL term or another related OCL project.

2.3.1 EMF

As previously mentioned, EMF, an Eclipse open source project, is a Java modeling framework which support code generation facility based on structured model [BSM+03]. Moreover, the mechanism in which it provides, exploits the customizable and extensible outcome. This could simply means that EMF keeps the generated code synchronized against the customization of the model, when it needs to be adjusted to the current requirement.

Initially, EMF implementation is based on Meta Object Facility (MOF) [Obj06a], an OMG standard for model driven engineering. As a consequence, EMF utilizes MOF-like core metamodel, called Ecore which is comparatively aligned with Essential MOF, part of OMG's MOF 2 specification. EMF supports several ways of getting the model into Ecore structure, which is identified as model importer. Currently there are 3 types of model importer available [DHMS07]: Java interface, UML models expressed in Rational Rose files, and XML schema.

Once the Ecore metamodel has been defined, Java implementation code including user interface can easily be generated. Moreover, EMF implements the EMF.edit framework which is useful in providing functional viewers and editors for the model. The generated code supports the standard operation for example, create, retrieve, update, and delete operations. The code can be regenerated repeatedly without worrying about the modification that has been made, because EMF implements JMerge which consistently keep the generated code synchronized with the preceding code.

For runtime supports, EMF also provides notification, customizable persistence to XMI, and reflective API features. Each EMF generated class will send the notification event whenever an attribute or reference is modified, as in observer design pattern [ERRJ95]. To persist objects, EMF employs a default XMI serializer. Furthermore, it can be extended by implementing custom serializer to write the code in any persistent form. Besides as an alternative way to read and write model, the reflective API of EMF can be used to manipulate instances of dynamic, non-generated, class.

EMF Code Generation

In general, EMF can generate Model implementation code, Edit framework code, and Editor framework code.

Model implementation code consists of interface and implementation for each modeled class. This implementation code provides the factory, as in the factory design pattern [ERRJ95], to create instances of model objects. Package class provides convenient code to access its metadata. The other features that the generated code provides include in switch utility, adapter factory base, and validator.

Edit and editor framework code shape the EMF.edit API. It is divided into 2 elements because they have different dependencies. Edit code, UI independent code, consists of item providers and item provider adapter factory. While Editor code, an UI dependent code, consists of model creation wizard, editor, action bar contributor, and advisor for RCP. Mainly the purpose of this framework is to provide the visual editor to manipulate the model in easier way.

The outline of the generation process starting from the beginning can be observed in the figure 2.3. It begins with the model which can be defined in XML schema, UML model (as Rational Rose file), or Java interface with the annotation. Using importer provided by EMF, each original model is converted to Ecore model. The other way to get the model into Ecore model is by creating the Ecore model directly either by provided Ecore editor, or with Emfatic [Dal05]. EMF can only generate code as long as the Ecore model is present. GenModel is the decorator, as in Decorator

Design Pattern [ERRJ95], of Ecore model which is built after the Ecore model has been defined. Primarily it is used to store the information which related to generate the model. One example of such information is the option that can be configured relevant only to the code generator, such as what packages to use, and where the generated code should go. Then with the JET template [Pop04] which contain the template of each generated code, GenModel has the capability to transform the model into java code. To overcome the synchronization problem between previous generated code and the newly one, this case happen when the generation process is iteratively made which would be the most likely case, JMerge will merge it into the correct synchronized code. The previous codes which do not want to be overwritten should be marked with "@Generated NOT" for each generated construct in its annotation. Otherwise, JMerge will overwrite it with the newly generated code.

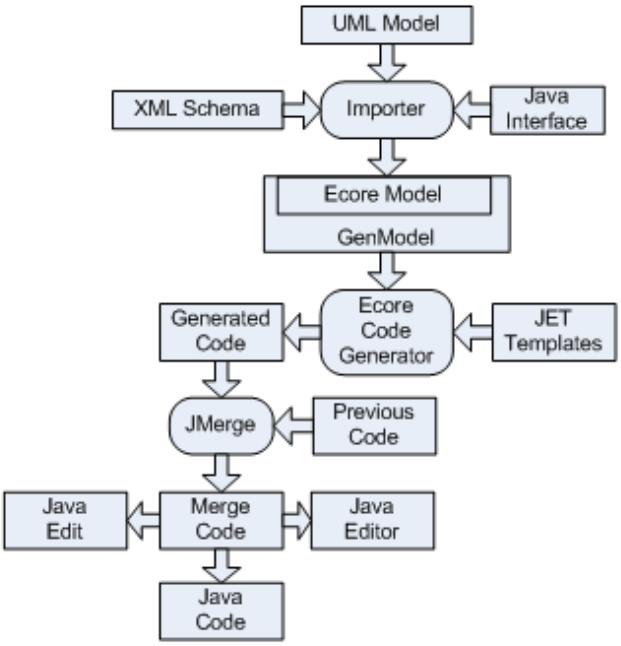

Figure 2.3: Outline of EMF Generation Process

Ecore and GenModel

Ecore is core model of EMF model, EMF metamodel. Since this implementation is based on MOF specification, Ecore can be seen as a subset of MOF-like core meta model.

One of the powerful elements of Ecore is EObject. All modeled objects within EMF implement EObject interface. It can be compared to Object in Java in some ways. Most of the EMF's fundamental features are provided by EObject. The reflective API, which introduces the eGet() and eSet() method to introspect object, includes in EObject features. Furthermore, EObject extends the Notifier by which any EMF model elements can be observed by sending notification when the state is changed.

The complete hierarchy of Ecore metamodel can be viewed in figure 2.4. It is taken from EMF Documentation [Tea06]. The detail of Ecore relationship diagram can be found in figure A.1 in appendix A.

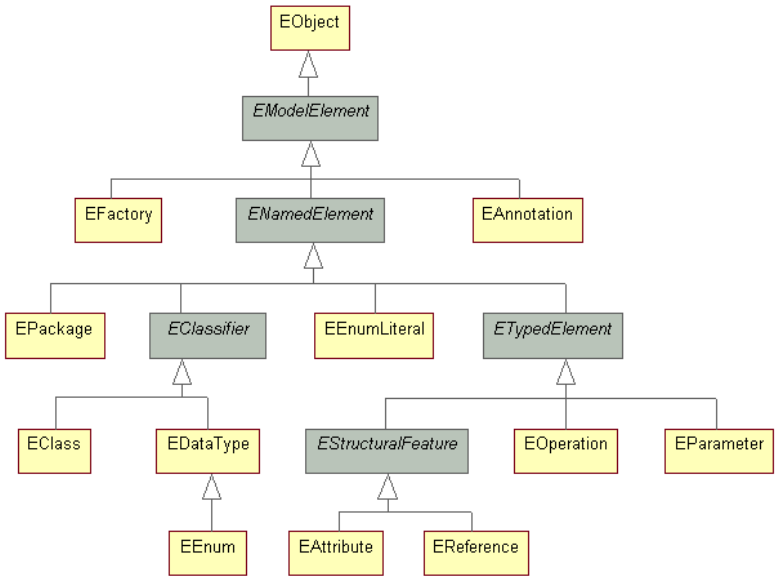

Figure 2.4: Hierarchy of Ecore diagram [Tea06]

Genmodel is a wrapper of Ecore model. It behaves as decorator to the Ecore model. It separates the information which needed specifically to generate the model. The information specific to the model remain in Ecore model, while the information which specific to model generation is stored in GenModel. This pattern maintains the high cohesion of the framework code. The EMF generator takes the GenModel instead of the Ecore model as the input.

EMF Generics

Java 5 has been included in the latest version of EMF (EMF 2.3) implementation. The interesting feature of Java 5 which in fact relevant with OCL compilation is Java Generics [Bra04]. The initial purpose of incorporating this feature was to generate typed list for multi valued features. In spite of this, along the way to achieve that, EMF team added new constructs that allow new java 5 concepts to be modeled directly in Ecore [MP07].

As we can see in the figure 2.5, the new classes which are introduced to support the Java Generics implementation in Ecore consist of ETypeParameter and EGenericType. ETypedElement, EClass, and EOperation now have been extended to have additional reference to EGenericType, besides their own prevous type reference.

The type parameter of EGenericType is represented by the reference to ETypeParameter, i.e., it can represent, T or E. The bounds of type parameter is demonstrated by eBounds reference of ETypeParameter to EGenericType, i.e. it can represent ⟨T extends A & B⟩. The raw type of EGenericType, represented as eRawType reference, is implemented in order to represent the erasure of the generic type.

The EGenericType has 3 references which refer to itself: eUpperBound, eLowerBound, and eTypeArguments. The reference's name is very well self-explained. All of the references represent upper bound, lower bound, and type argument respectively.

2.3.2 Eclipse OCL

Eclipse OCL is an implementation of OCL for EMF based models. It provides the basic infrastructure for parsing OCL constraint, specifying OCL queries and conditions,

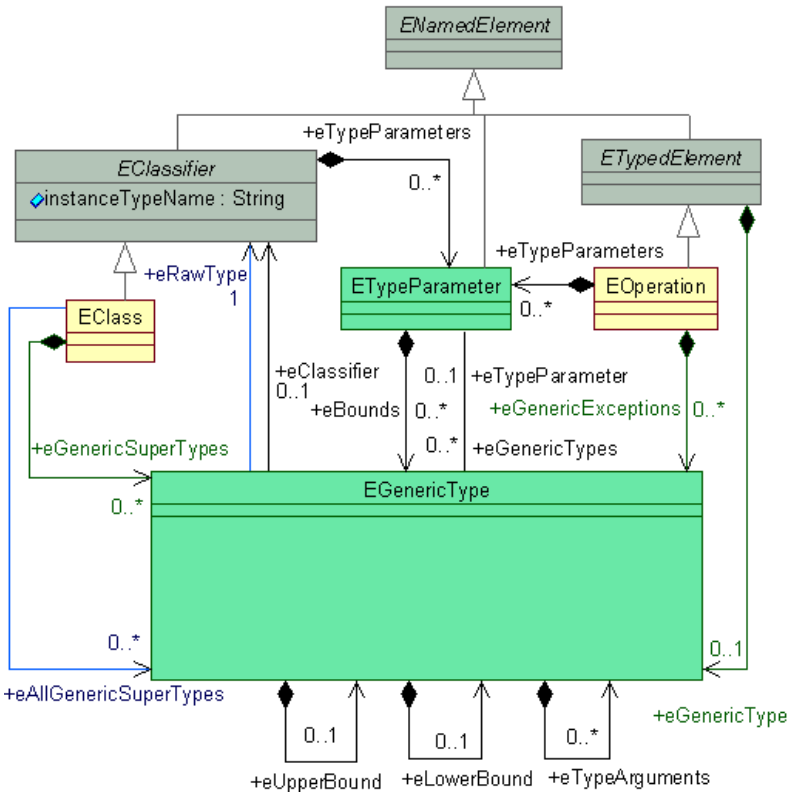

Figure 2.5: Ecore Generics [Tea06]

and validating as well as evaluating OCL constraint. It also defines the OCL abstract syntax trees (AST) to represent the OCL expressions. The AST representation can be inspected using the functionality provided by Visitor API. At the end it supports for serialization of parsed OCL expression.

OCL expression must be parsed before it can be processed in EMF. Eclipse OCL provides OCLHelper interface to parse the expression. This interface is presented in org.eclipse.ocl.helper package. It is a utility object provided by Eclipse OCL which provides a convenient API for parsing OCL expression.

The parsing process is wrapped in the creation of constraint method. This creation method is targeted to each type of OCL function expressions i.e. Invariant, Derive, Initial, etc, by defining method createXXX(), in which XXX refer to types of function expression. Those methods take expression string as the parameter. To create general constraint, the general method is used instead, which is named to createConstraint() method. Having the constraint parsed, it invokes the validation visitor to validate the expression. At the end, it returns the corresponding Constraint object if it succeed, otherwise it throws ParserException.

There are 2 possible reasons why the exception is thrown in parsing the expression [Tea07c]: the syntactical problems and contextual problems. The former includes mistakes in providing a proper syntax such as closing parentheses, keywords, type expression, etc. And the latter includes mistakes in providing context such as missing context, and wrong context, etc.

Once the constraint has been specified, parsed, and validated, it is ready to be further processed. Eclipse OCL provides the more interesting facility after wards. The defined constraints can be evaluated in order to get the results against the specified

object model in which it defines. All those tasks are encapsulated in very convenient facade defined by Eclipse OCL named OCL. It is defined in org.eclipse.ocl package.

The OCL expression is modeled using EMF. The result of parsing process is an AST of the OCL expression. The evaluation mechanism which is similar to aforementioned validation mechanism, follows the technique described in Visitor pattern [ERRJ95]. In evaluation case, the visitor visits all the element of AST of given constraint input, and do the appropriate evaluation task against the corresponding element. Eclipse OCL has provided Visitor interface which is inherited by the EvaluationVisitor to implement the specific operation of evaluating the OCL expression.

At the end, OCL expression, in AST representation, can be serialized to XMI file and deserialized it back when it is required. With this functionality, we can examine the parsed constraint resulted from Eclipse OCL parser.

2.4 Octopus

Octopus [Jos06] is one of the few of OCL tools currently available, which support the use of OCL. Octopus stands for OCL Tool for Precise UML Specifications. Octopus has the capability to check OCL expression syntax statically including the expression types and the correct use of the model elements in relation with defined OCL expression. The other potential feature of Octopus is the ability to transform UML model together with OCL expression into Java code. The combination of those capabilities gives the maximum result within OCL processing. While the syntax of OCL expression has been checked at compile time, the further process will be easier to carry out.

Octopus is designed as Eclipse plugins. To run it, Octopus requires to be installed in Eclipse platform. The generation mechanism provided by Octopus is rather straightforward. Figure 2.6 shows how the generation process from the model into java code is executed. Octopus model is formed by combination of UML model which is imported from XMI file and OCL expression. Once the Octopus model is accessible, Octopus code generator can perform the generation process.

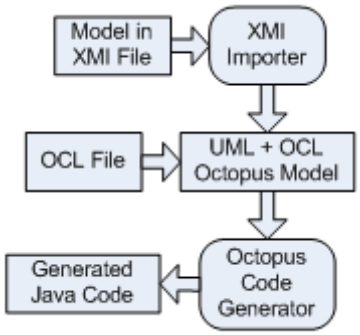

Figure 2.6: Outline of Octopus Generation Process

Royal and Loyal is one of the study cases of Octopus project which is extensively used in this work. It mainly models a computer system in a fictional company. The Royal and Loyal model which is represented in class diagram can be found in figure B.1 in appendix B.

2.5 Summary

In this chapter we have reviewed that UML model is not sufficient in representing the model. Therefore additional constraint is required. OCL is one of the constraint language which fits into such situation. It is precise yet understandable. This chapter also pointed out the benefit, the characteristics and the various use of OCL.

Beside that, the brief explanation around Eclipse Project which has relevancy with the OCL implementation, has been introduced. The top level project is Eclipse Modeling Project (EMP) which includes EMF and Eclipse OCL.

Furthermore, the introduction of Octopus, another tool which already supports OCL implementation, ends this chapter.

Chapter 3

Design

3.1 Design decisions

Before we jump into the architecture design of the compiler, there are some design considerations which needs to be taken into account. Following discussion gives an adequate amount of information about the decision and assumption made to this project.

3.1.1 Implementing OCL in Java

We have identified 2 techniques of implementation solutions in compiling OCL expressions into Java. We believe that there are more approaches which can be discovered. First approach includes in creating an operation for each OCL expression and creating another operations for the sub expressions if any. Another way is to create only one operation for each OCL expression and the sub expressions.

Octopus [Jos06] has implemented the first approach. An operation is created for each OCL expressions and for each sub expressions. Thus, the main operation might invoke the other operations which representing the compiled sub expression. Following is the example of this implementation.

```
context Customer
inv:cards->select(valid=true)->size()>1
```

The OCL expression defines an invariant of the Customer. It restricts the number of valid card of the Customer, which must be greater than 1. The result of Java implementation can be viewed in listing 3.1.

Listing 3.1: Octopus Java Implementation Code

```
public void invariant_Customer1() throws InvariantException {
  boolean result = false;
  try {
    result = (select11().size() > 1);
  } catch (Exception e) {
.
.
}
private Set select11() {
  Set result = new HashSet();
  Iterator it = this.getCards().iterator();
  while ( it.hasNext() ) {
```

```
      CustomerCard i_CustomerCard = (CustomerCard) it.next();
      if ( (i_CustomerCard.getValid() == true) ) {
        result.add( i_CustomerCard );
      }
    }
    return result;
}
```

From the listing, we can see that 2 operations have been created. The main operation is invariant_Customer1() which is a result of compiling the main OperationCallExp with the ' >' operator. The separate operation, select11(), is created as a result of representing the select operation which is a sub expression of the Invariant. The main operation invokes the sub expression operation to refer to the compiled result of its sub expression.

As oppose to that approach, the implementation only requires in creating one operation for each of OCL expression. The representation of the sub expressions, if any, can be alternatively implemented by defining the attributes and assigning the sub expression representation to the corresponding attributes. Instead of invoking the sub operation, the main operation uses the previous defined variable to obtain the sub expression. This implementation is expressed in listing 3.2.

Listing 3.2: Alternative of Java Implementation Code

```
public void invariant_Customer1() throws InvariantException {

  Set select11 = new HashSet();
  Iterator it = this.getCards().iterator();
  while ( it.hasNext() ) {
    CustomerCard i_CustomerCard = (CustomerCard) it.next();
    if ( (i_CustomerCard.getValid() == true) ) {
      select11.add( i_CustomerCard );
    }
  }

  boolean result = false;
  try {
    result = (select11.size() > 1);
  } catch (Exception e) {
.
.
}
```

Even though there is a possibility of additional required operation in representing OCL expression using the latter approach, for instance in the case when the inner expression depends on the outer expression, it significantly reduces the number of created operation. This condition results in the ease of maintaining the code in the future. Those considerations made us to decide on the latter alternative to implement the java code.

3.1.2 Implementation Strategy

Each function type of OCL expression, i.e., Def, Derive, Init, etc, are the specification defined by the standard. The specification states nothing about how it can be implemented to the specific programming language. In general, implementing the specification, the implementer attempts to find the mechanism in the target language which maps the same behavior with the specification. This is reflected when the

implementer aim to implement Def attribute expression in Java, which is defined as a getter method. In this case, the behavior of the getter method maps exactly with what the specification ruled about Def attribute expression.

In our case the situation is slightly different. We attempt to implement the OCL expression within EMF which already has a solid foundation in generating Java code. This compiler will make use the built-in block to provide the flexible solution. Therefore, for the Def case aforementioned, we can not implement easily the same behavior with getter method. One of the problem is the getter method has been generated automatically by the Ecore Code Generator. The previous solution, implementing Def attribute with getter method, will results in 2 getter methods generated in the final code.

Another tricky case is in implementing Init expression. The common implementation of Init expression is to initialize the corresponding element in the constructor. In EMF there is no custom constructor are created using their Code Generator functionality.

With those considerations, we decide to uniformly create an operation for each function type of OCL expression, for example defXXX() operation is created instead the getter getXXX() for Def attribute expression, initXXX() operation is created to compromise with the Init expression, etc. Nevertheless, this ad-hoc decision does not result in the way that the specification required. With the time constraints that we have, our concern now is more to the compilation results of the OCL expression itself. Given that the translation of OCL expression to Java expression correctly performed, the detail of that behavior implementation can be modified easily afterward. After the modification takes place, the implication, for instance, would be to remove the particular attribute to avoid 2 getter methods generated.

3.2 OCL Compiler Architecture

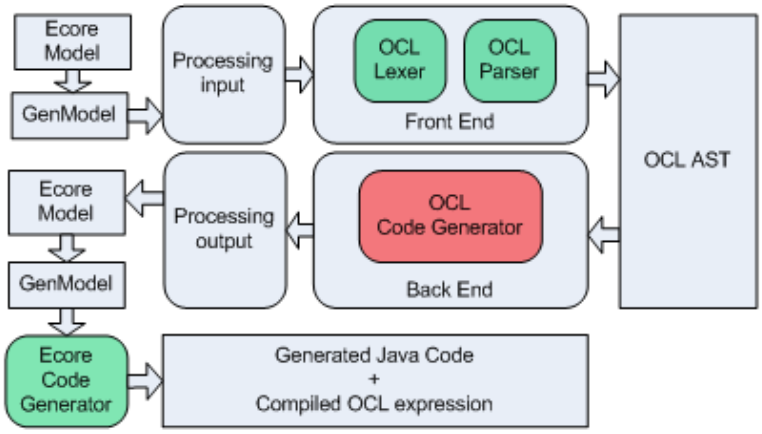

Figure 3.1: OCL Compiler Architecture

The general architecture of OCL compiler, illustrated by figure 3.1, is built within EMF. Some blocks of the compiler are supported by Eclipse OCL and EMF, marked with green box. The decision to take the GenModel as input is based on the nature of the GenModel itself. It is created in order to generate model out of it. The front end of the compiler has been provided by Eclipse OCL functionality. It performs lexing and parsing process to the OCL input. The output of this process, the AST of OCL expression, is consumed as input for OCL code generator, marked with red box. The

result of Back End process would be in the Ecore form. Ecore Code Generator then transforms the GenModel derived from previously generated Ecore into Java Code. The compiled OCL expression is included in the final generated Java code.

Each rounded box of the figure represents process while the box denotes the input or result of the process. The detail process of the whole compiler architecture is explained in the following subsection.

3.3 Processing Input

As of now, there is no definite way to declare OCL expression in Ecore model. Considering that, the approach to state OCL expression in Ecore has to be generic and flexible in a way that user can simply access it. In line with that, Ecore has the EAnnotation interface in order to provide mechanism by which additional information can be attached to any other core model object [BSM+03]. Referring to Ecore Relationship diagram in figure A.1 in appendix A, EModelElement has a reference to EAnnotation named eAnnotations with multiplicity many. Based on all those information, we decide to define the OCL expression as EAnnotation in Ecore.

As to the prior problem, the convention for stating the OCL expression has not been made within Ecore model. Nevertheless, in [Dam06], Christian W. Damus gave an example to specify OCL Expression in EAnnotation. Following that, table 3.1 lists the convention that is used in this OCL compiler project.

OCL Type	Attribute	Value
Def	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/define"
	Key	"def"
	Value	defExpression
Init	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/init"
	Key	"init"
	Value	initExpression
Derive	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/derive"
	Key	"derive"
	Value	deriveExpression
Invariant	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/invariant"
	Key	"invariant"
	Value	invariantExpression
Pre	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/pre"
	Key	"pre"
	Value	preExpression
Body	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/body"
	Key	"body"
	Value	bodyExpression
Post	Source Annotation	"http://www.eclipse.org/OCL/1.0.0/post"
	Key	"post"
	Value	postExpression

Table 3.1: Input Convention

With this format, each OCL expression is attached on corresponding context element. It supports several expressions in one context as the specification stated, for example any combination of multiple pre or post condition declaration in one context operation.

Processing input here means sorting out the OCL expressions from the model

which is attached in EAnnotation and then passing it to the next step. The order of which the input is processed is important because the semantics of the expression. Def expression must be processed before the others, while the other type of expressions might use that definition in their expressions. With that reason, the necessity of processing Def expression in this stage is only to define the particular element, either EAttribute or EOperation, without necessarily process the definition expression, the right side of Def expression.

3.4 Front End Compiler

The front end of compiler analyzes the input to build an internal representation, known as Abstract Syntax Trees (ASTs), of OCL expression. This phase is divided into 2 tasks, Lexical Analysis and Syntax analysis. The former breaks the input into small pieces called token which is already defined for OCL expression. And the latter parses the token sequence to identify the correct syntactic structure based on OCL specification.

Those tasks have been encapsulated by Eclipse OCL in HelperUtil which presents in org.eclipse.ocl.internal.helper package. It is a utility class which support of the creation and implementation and of the aforementioned OCLHelper.

3.5 Back End Compiler

The responsibility of this building block is to transform OCL ASTs to Java code encapsulated in Ecore model. EMF has already provided Ecore code generator which transforms the model represented in Ecore to Java code. In this design we utilize this feature to generate Java code. Therefore, the outcome of this block should be an element of Ecore which can be customized in the EMF code generation to get correct generated code.

In generating source code, EMF exploits JET template [Pop04] to reduce the complexity and increase the readability and extensibility of the program which generates code. Using the extensibility offered by the template, modeler, for instance, can provide body of the operation with a certain construction, and then let the EMF code generator generates the code. This strategy fits with the requirement of this OCL compiler. Each translated OCL expression of this building block can be passed as an input for the template which later will be generated to java code. Following subsection will explain more about the detail within this building block.

3.5.1 Code Generation Strategy

We have identified 2 alternative approaches, among others, in generating Java code out of OCL ASTs. First approach, we could perform direct translation from OCL ASTs to Java code. While in the second approach we can introduce the intermediary model to represent the OCL ASTs in which in the later phase the second transformation performs the translation to valid representation of Java code. The former option might be better in level of performance as it requires less overhead task than the latter. However, the second alternative provides better extensibility in the future and results in much higher cohesion generating code which in turn makes easier to maintain. Considering that, we decide to set the intermediary model to represent the OCL ASTs as an intermediate step before getting into real Java code, as illustrated in figure 3.2. The intermediary model is called Imperative Ecore.

The design consideration of Imperative Ecore model is based on the characteristic of OCL expression and the generated Java Code. As previously explained, OCL is a type language. It implies that each OCL expression has a type. To complete

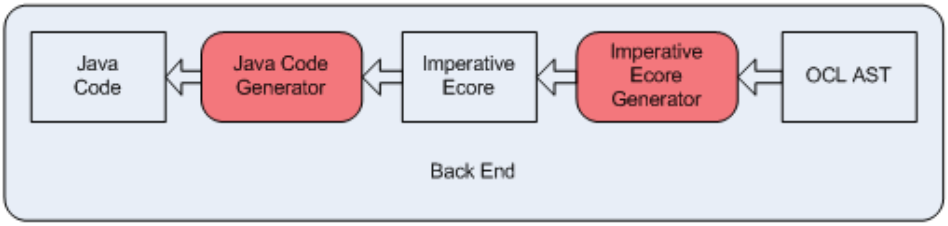

Figure 3.2: OCL Compiler Back End

the translation process, the compiler has to transform the OCL expressions together with the OCL types into Java representation. The idea of Imperative Ecore model is to represent OCL ASTs in Java ASTs without converting its OCL types. The OCL types remain in Imperative Ecore representations. The discussion regarding Imperative Ecore model will be presented in subsequent subsection.

The first generation process is to generate OCL ASTs to Imperative Ecore model. To complete the generation process, second step is required. It transforms the Imperative Ecore model to Java expressions. Note that, the Java representation at this level will be encapsulated in Ecore model, which means that all representations are EModelElement element, refer to Ecore Hierarchy in figure 2.4. The Imperative Ecore model and the generated Java expressions will be stored as an element of Ecore model, for instance, EAnnotation.

3.5.2 Imperative Ecore Model

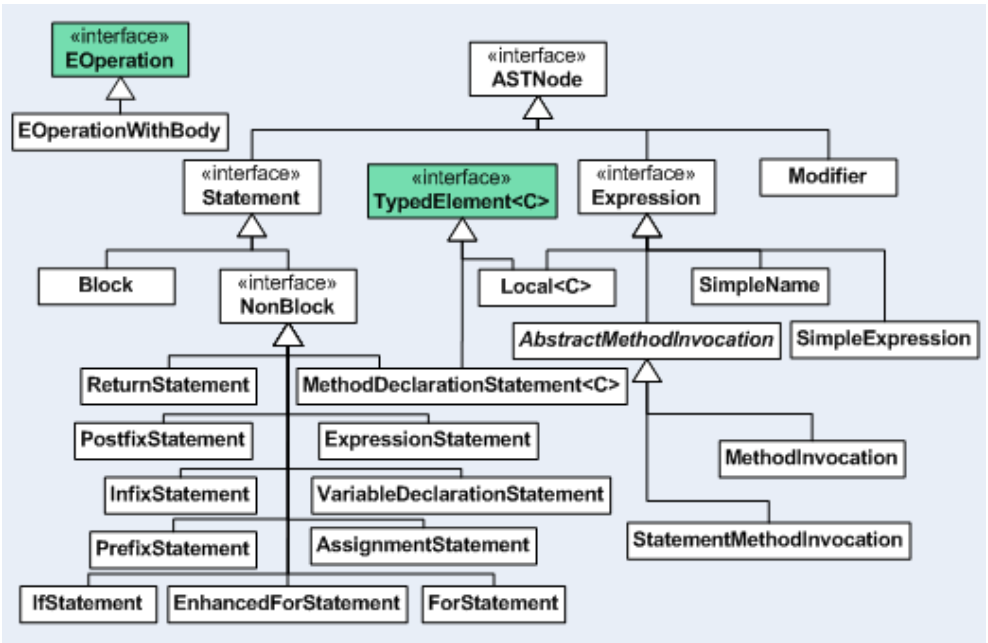

Figure 3.3: Hierarchy of ImperativeEcore

The design of imperative Ecore model is based on Java ASTs with additional functionality. The purpose of this intermediary model is to provide a representation of OCL expressions in Ecore with OCL types instead of Ecore types. It consists

of 3 packages: statement, expression, and utilities. All classes are inherited from utilities.ASTNode, except utilities.EOperationWithBody. ASTNode functions as an umbrella for those classes. EOperationWithBody subclass of EOperation of Ecore model instead. The hierarchy of ImperativeEcore Model is ilustrated in figure 3.3

The general idea of this model is to have a representation of Java statements. This is achieved by creating Block and NonBlock element which is inherited from Statement interface. NonBlock represents the single statement, while Block is composed of several statements. All possibility kind of statement which could be derived from OCL expression inherits the NonBlock, such as, IfStatement, AssignmentStatement, ForStatement, etc. Most of the constructs is originated from Java AST with slight modification when the java type is involved.

In Ecore, Java type is represented by EClassifier. An Object which holds this kind of property, must inherits ecore.ETypedElement, which has eType reference to EClassifier. In ImperativeEcore, as we design it to preserve OCL type instead of Ecore type, such object inherits ocl.utilities.TypedElement<C> provided by Eclipse OCL. Local<C> and MethodDeclarationStatement<C> of ImperativeEcore model elements implement TypedElement<C> as illustrated in the hierarchy 3.3. Those classes represent the local variable and method declaration respectively which preserve the properties of OCL types. The C type parameter will later be bound to EClassifier in the case of Ecore implementation.

Following is the example of ImperativeEcore representation of given Java code. Listing 3.3 expresses the coll() operation which defines the collection creation and assignment.

Listing 3.3: Example Java Code

```
private Collection<Integer> coll() {
  Collection<Integer> result = CollectionUtil.createNewSet();
  result.add(1);
  result.add(2);
  result.add(3);
  return result;
}
```

And the ImperativeEcore model representation follows:

```
<MethodDeclarationStatement type="Sequence(Integer)" name="coll">
   <Modifier name="private"/>
   <Block>
      <VariableDeclarationStatement>
         Sequence(Integer) result = CollectionUtil.createNewSet()
      </VariableDeclarationStatement>
      <ExpressionStatement>result.add(1)</ExpressionStatement>
      <ExpressionStatement>result.add(2)</ExpressionStatement>
      <ExpressionStatement>result.add(3)</ExpressionStatement>
      <ReturnStatement>result</ReturnStatement>
   </Block>
</MethodDeclarationStatement>
```

Another important issue in ImperativeEcore is the present of EOperationWithBody. It is the only class which does not inherit from utilities.ASTNode. The idea of EOperationWithBody is to provide the representation of EOperation which includes the body of corresponding operation. The declaration of the body is established by providing reference to statement.Block.

3.6 Processing Output

At this level the compiled OCL expression has been generated. It is stored in EAnnotation of corresponding EOperation of Ecore model. The remaining task is to serialize the Ecore Model contained all the compiled expressions. This activity can be achieved by utilizing Ecore serializer provided by EMF. The further required information regarding serialization to Ecore file can be obtained from GenModel.

EMF code generator will generate Java code out of the GenModel derived from the serialized Ecore model. At the end, the compiled OCL expressions is included in the generated Java code encapsulated in the corresponding operation.

3.7 Summary

In this chapter, the design decisions and the architecture of the compiler has been clearly described. Figure 3.1 provides the depiction of the OCL compiler architecture. Generally Eclipse OCL and EMF features have been contributed and included in the design process.

The OCL compiler takes a GenModel as an input. This GenModel is derived from the input Ecore model in which OCL expressions are attached as EAnnotation of the appropriate context element. After processing the input, including define the property and operation for each Def Expression encountered, Eclipse OCL takes place to perform the front end compiler task, i.e., lexing and parsing the input OCL expressions. The result of this process is OCL ASTs which will be passed to the back end compiler. In the back end compiler, the process is further split. The first one is to transform the OCL ASTs to ImperativeEcore model as an intermediary model. The second process is to generate the Java expressions out of the ImperativeEcore model. The serialization of Ecore model into Ecore file finalizes the compilation process. At the end, generated Java code included compiled expression is obtained from generation performed by Ecore code generator.

Chapter 4

Implementation

All the implementation code of this OCL compiler is developed under Eclipse platform. Therefore there are many patterns of Eclipse is used throughout the code in various place, for example the code for generating action in one of Eclipse's editor or the code for extending another plugin as an extension point. All the framework architectures provided by Eclipse already adhere to some well known design pattern such as Factory Pattern, Adapter Pattern, and many more, which makes the developer easier and faster in extending and customizing it.

4.1 OCL Compiler Implementation

Class diagram of the primary element of the implementation is shown in Figure 4.1. ExpressionGenerator class plays as a central role, the main class of this OCL compiler, which takes the GenModel included the OCL expressions as an input.

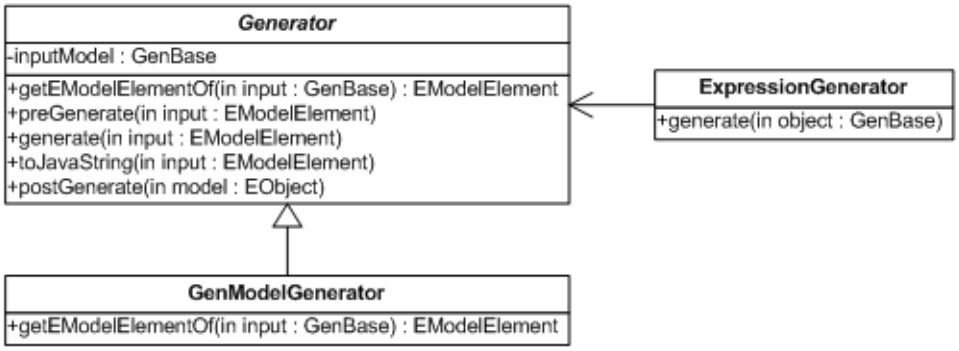

Figure 4.1: Generator Class Diagram

ExpressionGenerator delegates the compilation function to Generator abstract class. The subclass of Generator determines which subtype of input that it can accept. Currently there is only one class inherits the Generator abstract class, named GenModelGenerator. It is possible to extend the generator functionality by subclassing it with another generator implementation class. Nevertheless it might break your model consistency when it is not used properly.

GenModelGenerator is provided in order to work with GenModel type input. Additional subclasses of Generator might introduce other forms of input, as long as it still conforms to GenBase hierarchy. Note that the type of inputModel property of

Generator abstract class is GenBase. Therefore all the subclasses have to deal with GenBase type of this property. Figure 4.2 shows the hierarchy of GenBase.

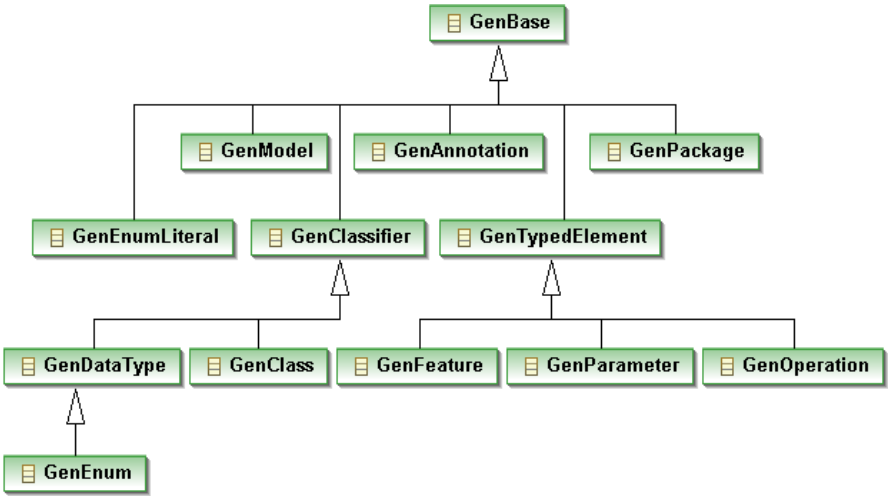

Figure 4.2: Hierarchy of GenBase

Moreover, all the subclasses of Generator has to provide getEModelElementOf() operation, since it is defined as an abstract. This operation returns the Ecore model element of the GenBase input as EModelElement type which will be further processed. EModelElement is used to represent the model element in Ecore. The hierarchy of EModelElement in Ecore model can be recalled in Ecore hierarchy in figure 2.4.

This Generator architecture is originated from Ecore Code Generator. At present they provide the generator for GenClass, GenEnum, and GenPackage beside the generator from GenModel.

There are 4 main methods of Generator abstract class provided in this implementation: preGenerate(), generate(), toJavaString(), and postGenerate(). Those methods are invoked from generate() method of ExpressionGenerator class. All those methods perform the complete compilation process of this OCL compiler implementation. The subsequent subsection explains the detail of each of the operation process.

4.1.1 PreGenerate

preGenerate() method initializes the condition before the compilation takes place. In our case, it process the Def expression as explained in section 3.3. Thoroughly, it finds all the Def expression which is attached on EAnnotation of input model. For each Def expression encountered, the Eclipse OCL front end compiler parses the expression and yields in the corresponding element of definition. The newly defined element afterward has to be linked to its owner class. Note that, in this step the Def expression has not been generated yet. The preGenerate() method does the parsing and yields the corresponding element, i.e Eattribute or EOperation and links them to the owner class. The Def expression which still has to be applied on particular element will be processed together with the other expressions in the next step.

This process is clearly illustrated in figure 4.3. Input of EModelElement initially contains all types of OCL expressions represented as string in EAnnotation of the corresponding type element. DefVisitor visits the structures of input model to find and process the def expressions. The result is the creation of new EAttribute and EOperation of corresponding Def expression together with its EAnnotation represent the Def expression, which marks with yellow in the right block of the figure. Note

that at the beginning Def Attribute and Def Operation expressions are defined in EClass, but in the result, they are redefined in the newly created element for further processing.

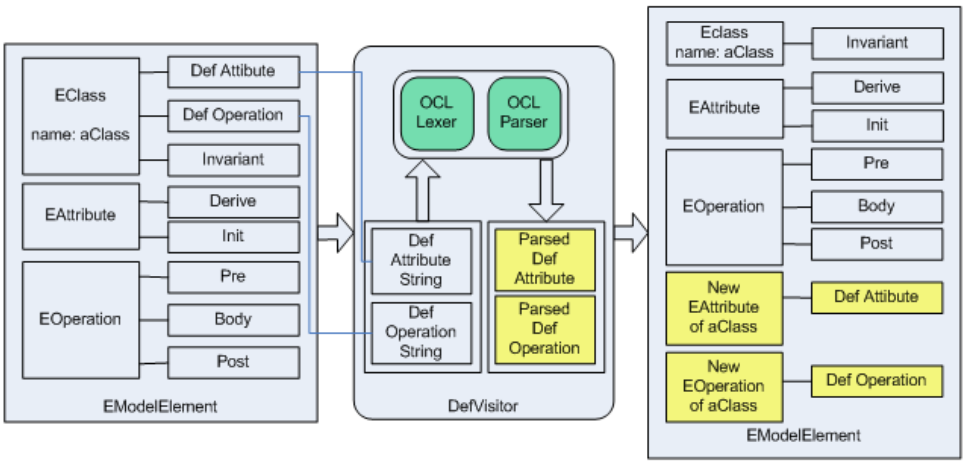

Figure 4.3: Process of preGenerate()

The approach in traversing the structured element i.e., Ecore model, is principally based on the Visitor pattern [ERRJ95] with an additional walker class to perform the visit order. The detail of the walker class implementation is explained in section 4.3. Mainly it performs visiting all the structures of input to find the Def expression and then it operates the parsing and linking action for each finding. In fact many of the implementation of this compiler follow this pattern. In this circumstance, the visitor class which has the responsibility to do the task is DefVisitor. The complete picture of the visitor architecture of this compilation process will be discussed in the next subsection.

4.1.2 Generate

The generate() method performs the real generation of function types of OCL expression, i.e. Def, Derive, Init, Invariant, Pre, body, and Post expression. To accomplish those tasks, 3 visitor classes have been developed. As a matter of fact, 1 visitor class is adequate to complete all the tasks, but to preserve the high cohesion in the design code, they are divided into 3 visitor classes: AttributeExpressionVisitor, OperationExpressionVisitor, and ClassifierExpressionVisitor. The division is based on, as its name suggests, part of the elements of the input that they visit. Each Visitor has its own responsible expression to encompass. AttributeExpressionVisitor visits all the expression which is attached in EAttribute: Def, Derive, and Init expression. OperationExpressionVisitor visits all the EOperation which eventually will find these expressions: Pre, Body, and Post expression. And ClassifierExpressionVisitor visits all the expression which is attached in EClassifier: Invariant expression. Figure 4.4 exhibit the hierarchy of Visitor which is used to realize the objective.

EcoreSwitch<T> is the superclass of those visitor classes. It is generated by Ecore Code Generator. It facilitates the subclass to have the capability to visit all the structure elements of the Ecore, i.e., EAttribute, EClassifier, EOperation, etc. AbstractExpressionVisitor is the abstract class which carries out the common task between the subclasses, for example the functionality to retrieve the OCL expression which is stored in EAnnotation. The DefVisitor has already explained in previous section 4.1.1. It performs preprocessing to define def expression. AttributeExpressionVisitor, OperationExpressionVisitor, and ClassifierExpressionVisitor are used for

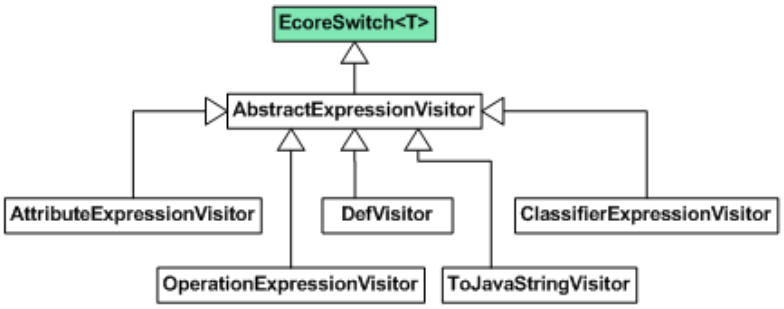

Figure 4.4: Visitor Hierarchy

main processing in compilation process. At the end ToJavaStringVisitor is exploited to finalize the process. It will be explained afterward.

The core compilation process is performed by the 3 main visitor classes: AttributeExpressionVisitor, OperationExpressionVisitor, and ClassifierExpressionVisitor. Those visitors visit the appropriate element of the model to find the OCL expressions which later will be compiled to ImperativeEcore representation. The execution of those visitors results in extended Ecore model. The corresponding element of the model has EOperationWithBody element, which includes the compiled OCL expression in ImperativeEcore representation, as its EAnnotation.

AttributeExpressionVisitor

AttributeExpressionVisitor provides the handler of EStructuralFeature element, named caseEStructuralFeature(). Providing such handler means that this visitor knows how to visit all EStructuralFeature element of input model which later can perform the required action against it. For each visiting, this visitor looks for OCL expression which is defined in EAnnotation of the element. Subsequently, OCL expression will be compiled into ImperativeEcore model. At the end of this process, the compiled OCL expression is transformed to EOperationWithBody in order to be translated to java expression in the next step. The whole process which is performed by AttributeExpressionVisitor is briefly illustrated in figure 4.5.

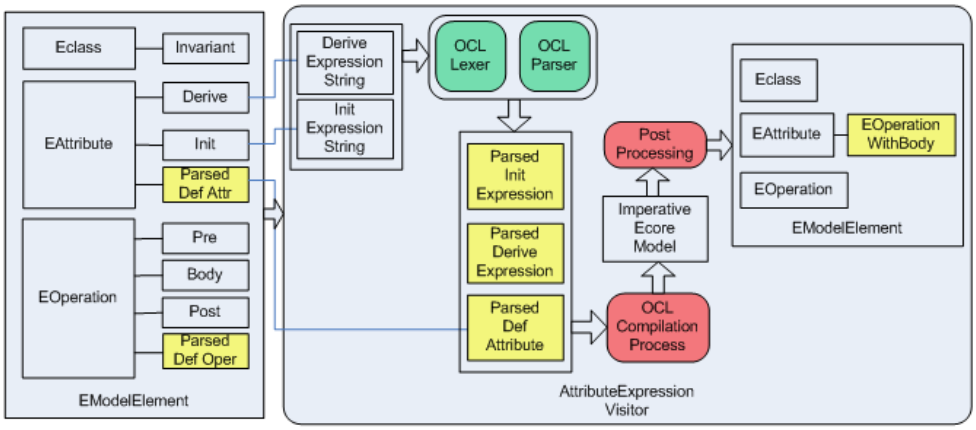

Figure 4.5: Process of AttributeExpressionVisitor

The possible type of OCL expressions that can be found in this element, EStructuralFeature, as previously explained in section 2.2.2, are Def attribute, Derive, and Init expression. Prior to further processing, those OCL expressions are required to be parsed. Note that from preceding phase, preGenerate(), we have already parsed the Def expression. Therefore, in this phase, only the remaining expressions need to be parsed by Eclipse OCL. We can see in the figure, the Def attribute of the input, marks with yellow which means it has been processed, skips the parsing process while the Derive and Init expressions have to go through the parsing process.

The parsed OCL Expression is eligible as input of the next compilation process. This process is primarily performed by another visitor, CompilationVisitor. It visits all the metamodel elements of OCL expression to perform the compilation. The detail of this process is explained in section 4.4. The result of that compilation is the compiled OCL expression represented in ImperativeEcore Model. Considering that the final result will be translated to java string performed by Ecore Code generator together with JET Template, the intermediate compiled expression will be stored in EOperationWithBody which is attached in EAnnotation of corresponding element. The visual representation of this can be seen in the last result shown in figure 4.5.

The following example is given to clarify all the described process in this step performed by AttributeExpressionVisitor. This example is based on The Royal and Loyal study case [WK03]. The Class Diagram of the model can be found in figure B.1 in appendix B. The OCL expression defines the 'initial' property of Customer class and is assigned to substring of 'name' property which already defined in the model.

```
context Customer
def:  initial:String = name.substring(1,1)
```

The intermediate compilation result of that expression is wrapped in EOperationWithBody which is attached in the newly defined element, EAttribute named "initial". Note that the name of the corresponding operation is defInitial() instead of getInitial(), as discussed in design section 3.1.2. The output below has been serialized into XML format.

```
<EAttribute name="initial" type="EString">
   <EAnnotation source="ASTNode/Def">
      <EOperationWithBody name="defInitial()" type="EString">
         <Block name="result">
            <ReturnStatement>this.getName().substring(1-1, 1)
            </ReturnStatement>
         </Block>
      </EOperationWithBody>
   </EAnnotation>
</EAttribute>
```

OperationExpressionVisitor

The compilation process performed by OperationExpressionVisitor is comparatively similar with the process executed by AttributeExpressionVisitor. This visitor visits all EOperation element by providing caseEOperation() handler. In this element, EOperation, there are 4 possibilities of the target OCL Expressions which include Pre, Body, and Post expression as well as Def Operation expression. Def Operation has been parsed in the previous phase, while the remaining expressions must go through parsing process. Having all the expressions parsed, they are compiled to ImperativeEcoreModel and followed by the creating EOperationWithBody elements. This process is illustrated in figure 4.6.

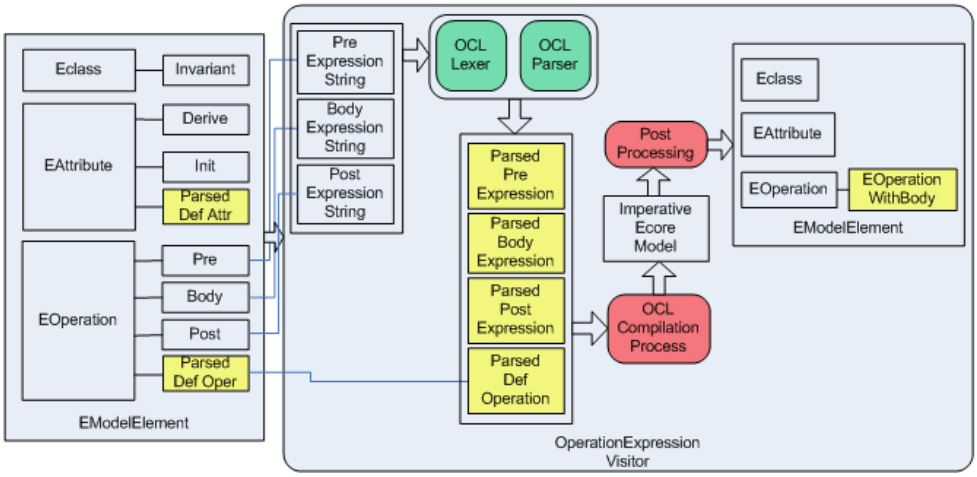

Figure 4.6: Process of OperationExpressionVisitor

ClassifierExpressionVisitor

ClassifierExpressionVisitor is another equivalent visitor which performs the compilation process of OCL expression which is keyed in EClassifier, i.e., Invariant expression. The whole process is similar except that it takes Invariant as the input expression. And the result of EOperationWithBody is attached on EClassifier. Figure 4.7 illustrates the process.

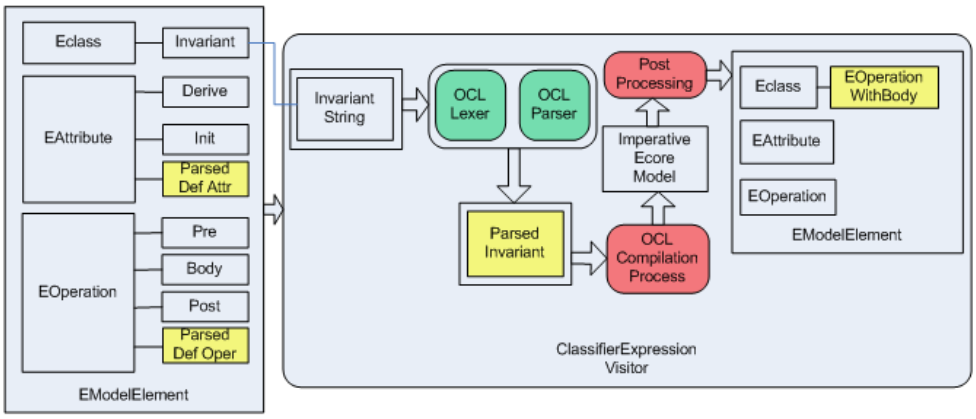

Figure 4.7: Process of ClassifierExpressionVisitor

4.1.3 ToJavaString

This operation generates Java string of the ImperativeEcore model resulted from preceding phase. To accomplish the task, ToJavaStringVisitor has been developed. This visitor visits all EAnnotation of the input element to find the EOperationWithBody which is attached as the content of EAnnotation. The main objective of this visitor is to generate the Java string out of the body of EOperationWithBody element which is still represented in ImperativeEcore model. The detail process of this operation is exhibited in figure 4.8.

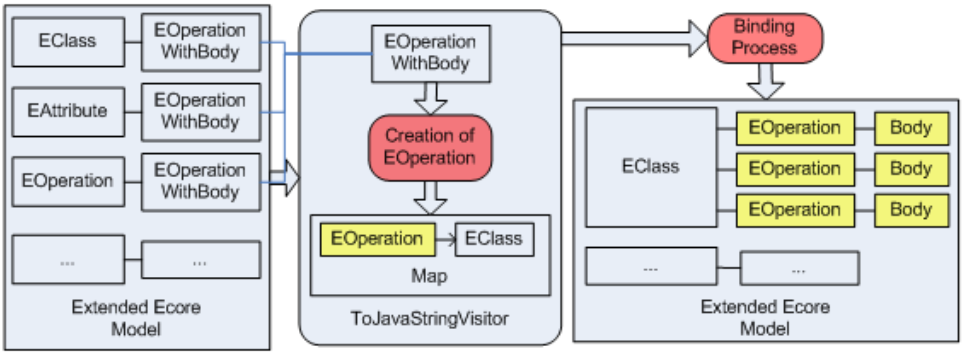

Figure 4.8: Process of toJavaString() operation

The utilities.ASTNode, the superclass of ImperativeEcore model, has already toJavaString() method embedded. It provides the Java string representation of each model element. Therefore, to generate the Java string, ToJavaStringVisitor can simply invoke toJavaString() method of each Statement encountered in body of EOperationWithBody, except that for VariableDeclarationStatement and MethodDeclarationStatament. At this point, all MethodDeclarationStatement has been transformed to EOperationWithBody. Therefore, only VariableDeclarationStatement remains in the model. Remember that, ImperativeEcore represents the Java statement model with the OCL type instead of Java type. In this case, part of toJavaString() of VariableDeclarationStatement will return OCL type string instead. This is not the case for the other elements of ImperativeEcore due to the fact that the type is not a constituent of their Java String representation. The translation of VariableDeclarationStatement element to Java string requires additional functionality which has the capability to map from OCL type to Java type. It will be discussed in section 4.2.

EMF code generator knows how to generate the body of the operation. Modeler has to specify the body in the EAnnotation of the particular EOperation with certain rules. The EAnnotation should have the source attribute with the value of "http://www.eclipse.org/emf/2002/GenModel". The detail entry of the EAnnotation should have the key with the value of "body", and the body of the operation is stored in value property. The following example which is in Ecore representation describes it in a clearer way.

```
<eOperations name="availableSeats" eType="...">
   <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
      <details key="body" value="body of the operation"/>
   </eAnnotations>
</eOperations>
```

According to the example, Ecore Code Generator accompanied by JET templates will create the operation named availableSeats with the specified type and fill the body with the value of value property of the details of its EAnnotation. In this situation, the value would be "body of the operation".

Referring to that, each EOperation is generated based on EOperationWithBody found in the corresponding element of the model, and store the generated Java string in its EAnnotation by using similar technique described. Afterward, Ecore Code Generator will generate correct java code out of it.

The process of creation EOperation based on EOperationWithBody is performed by ToJavaStringVisitor as shown in figure 4.8. The subsequent process is to bind the

newly created EOperation to its owner class. This process can not be done in visitor class while we can not change the model at the same time when it being visited. Therefore the process is shifted out of the visitor. At the end of this process, the complete Ecore model with the EOperation and the annotation body which represents the implementation of OCL expressions in Java code is ready.

4.1.4 PostGenerate

The objective of this part is simply to persist and serialize the Ecore model to ecore file. It is achieved by invoking the serializer code provided implementation by EMF. The information of where the destination file should be placed could be retrieved from GenModel. The code snippet 4.1 shows how the serialization is implemented.

Listing 4.1: Serialization code

```
.
.
URI uri =
    URI.createPlatformResourceURI(filePath + ".ecore", true);
Resource resource = new ResourceSetImpl().createResource(uri);
resource.getContents().add(ecoreModel);
resource.save(null);
.
.
```

4.2 OCL Types to Ecore Types

Eclipse OCL has utilized the generics feature of EMF, refer to section 2.3.1, in implementing OCL. At present the Eclipse OCL implementation has been widened to support not only for EMF implementation but also for UML2 [Tea07b] implementation. UML2 is another sub project of MDT, refer to section 2.3.

The power of generics is significantly used by Eclipse OCL implementation. Eclipse OCL defines all the model elements with necessary type parameter. The specific implementation, in this case EMF and UML2 will later provides the binding of type parameters to type arguments. With this way, both EMF and UML2 can use the generics implementation of Eclipse OCL. In this project, since we only concern to the specific implementation of EMF, we do not explain about the specific implementation of Eclipse OCL for UML2. In this report, the Eclipse OCL implementation for EMF will later be called OCL Ecore to avoid confusion with the generics implementation or the other specific implementation.

Eclipse OCL should represent all OCL types specification in their implementation. The specialized EMF model of OCL type, OCL Ecore type, is exhibited in figure 4.9. We can directly compare it with the one shown in figure 2.2 in section 2.2.1 which belongs to OCL Specification.

Eclipse OCL is originally implemented in order to provide the capability to evaluate the OCL expression at runtime. Therefore, the Eclipse OCL standard library, which holds the properties of the type elements, is also included in their implementation. While in Ecore compile time we do not preserve the OCL standard library anymore, thus we should transform the OCL Ecore types to Ecore types. To apply the OCL Ecore types within Ecore model, the mapping between Eclipse OCL types to Ecore types is required. For primitive types the mapping is straightforward. Table 4.1 shows how is the mapping implemented. Note that in the current version of OCL Ecore type hierarchy, subclass of PrimitiveType has been eliminated. However they provide name attribute to identify which kind of primitive type it represents.

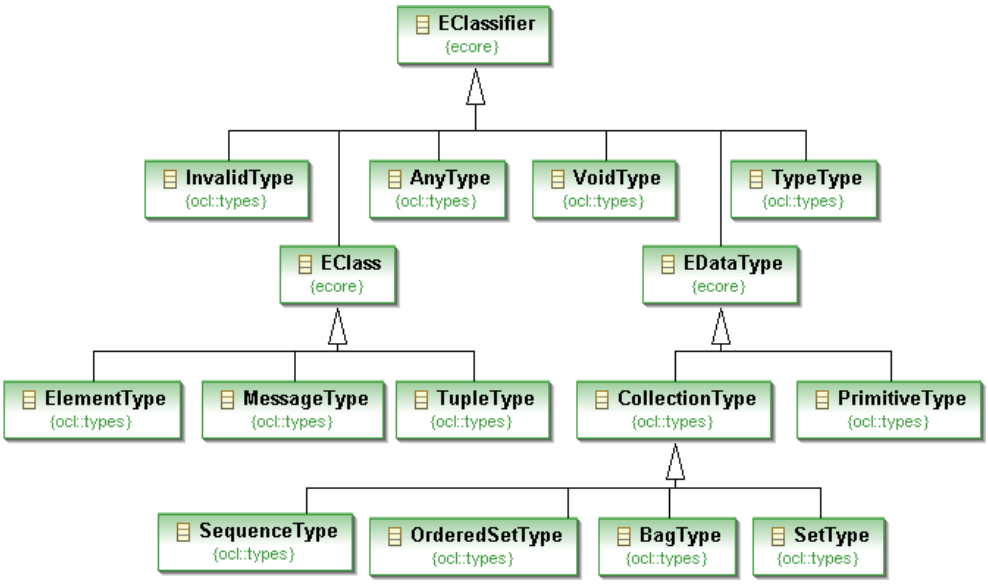

Figure 4.9: Eclipse OCL Types for Ecore implementation

In Ecore, list for multi valued features is represented in EList. EList subclass java.util.List. Thus, all CollectionType of OCL type should be transformed to EList. For the rest of OCL Types currently we support as EObject in Ecore type.

OCL Ecore Type	Ecore Type
PrimitiveType.Boolean	EBooleanObject
PrimitiveType.String	EString
PrimitiveType.Integer	EIntegerObject
PrimitiveType.Real	EDoubleObject
PrimitiveType.UnlimitedNatural	EDoubleObject

Table 4.1: Primitive Types mapping

4.3 Visitor Pattern with Walker

In our implementation, we introduce the additional class, known as walker class, to define the visit order of the structured element. This additional class is not required in the original visitor pattern while the visit order has already defined in the structure model itself. This walker class is required when the structure model which needs to be visited does not define the visit order. Ecore model has not expressed the visit order in their structure since it is not designed to be visited by Visitor Pattern. As an alternative it provides the switch class, org.eclipse.emf.ecore.util.EcoreSwitch<T>, by which the client can subclass it and provide the specific handler. Switch class behaves like the Visitor class. It provides the handler for each structured elements of the model. While the model itself does not provide any operation to define the visit order, the walker class can independently define how the visitor would traverse the model. With this approach, the visit order can be even more flexible as we can define it as it is required.

The type parameter T of EcoreSwitch is associated with the return value of the handler which it provides. The subclass has to provide the correct type argument as it is required. In general case, it can be substituted with Object. In that case, all return value of the handler provided by the subclass must conform with Object type.

In our case, to visit all elements within Ecore model, which is required in implementing the compilation process, we have defined the EcoreWalker class. It provides the visit order to traverse all the elements of Ecore model. There are 2 main methods in this class, as can be seen in listing 4.2. EcoreWalker is designed to traverse within the EModelElement type. Furthermore, it supports the EcoreSwitch<Object> to visit the model element. Those conditions are reflected on the signature of both operations.

Listing 4.2: EcoreWalker code

```
public Object walk(EModelElement eME,
                                    EcoreSwitch<Object> v) {
    v.doSwitch(eME);
    visitOwnedParts(eME, v);
    return v;
}
private Object visitOwnedParts(EModelElement e,
                                    EcoreSwitch<Object> v) {
    for (Object oa : e.getEAnnotations()) {
        EAnnotation a = (EAnnotation) oa;
        walk(a, v);
    }
.
.
```

This traversing order is simply defined by the first 2 lines of the walk() method. The first line invokes the handler of its own element, v.doSwitch(eME), and the subsequent line invokes the second method which defines the traversing order of part of its own element. In the visitOwnedParts() method, the definition of how the visitor should traverse the elements of the model should be clearly defined. At the end it returns some value. In this situation we pay no attention to the return value as we do not need it. Therefore any return value will do.

Listing 4.3 shows the invocation to traverse the Ecore model element. Input represents the EModelElement which will be visited. DefVisitor, referring to its hierarchy in figure 4.4, is one of the qualified visitor which can be accepted in walk() method.

Listing 4.3: Invocation to EcoreWalker

```
.
.
EcoreWalker ecoreWalker = new EcoreWalker();
ecoreWalker.walk(input, new DefVisitor());
.
.
```

4.4 OCL Compilation Process

This sub process is executed within the 3 core visitors previously discussed, i.e., AttributeExpressionVisitor, OperationExpressionVisitor, and ClassifierExpressionVisitor. The main goal of this sub process is to compile the parsed OCL expression into ImperativeEcore model representation. As previously explained in section 3.1.1,

the approach using one operation to implement OCL expression is adopted in this implementation. According to that, the result of this compilation process is statement.Block which contains a number of statement.Statement represents the body of corresponding operation.

4.4.1 CompilationVisitor Implementation

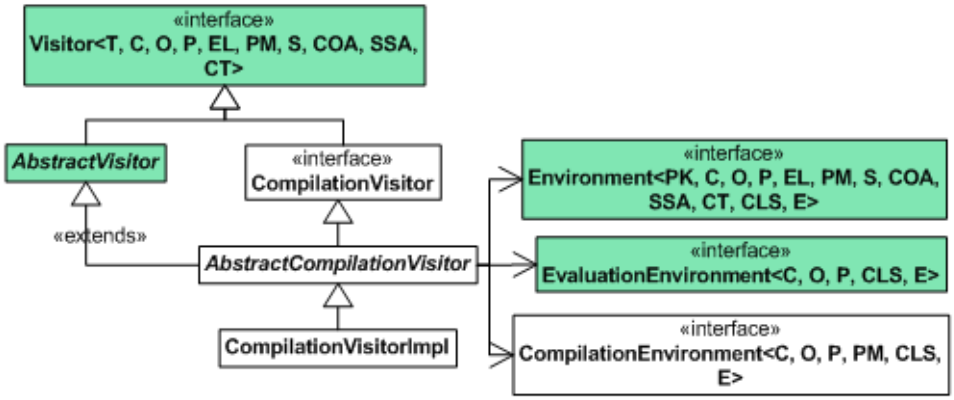

Figure 4.10: Class Diagram of CompilationVisitor

Performing the compilation process of AST of OCL expressions, another visitor has been developed, named CompilationVisitor. This visitor visits all the elements of OCL metamodel to perform the compilation which results in ImperativeEcore model. The class diagram of this visitor is provided in figure 4.10.

The superclass of CompilationVisitor is Visitor<T, C, O, P, EL, PM, S, COA, SSA, CT> which has been provided by Eclipse OCL. The visitor superclass has many type parameters. They are defined in order to give the generic implementation of visiting OCL expression. With those type parameters, the subclass can provide solution either to specific EMF implementation or to specific UML2 specification. The Visitor superclass provides the generic handler to all elements of OCL metamodel. The relation of type parameters depend on each handler. Example signature of generic handler provided by Visitor can be observed in listing 4.4.

Listing 4.4: Generic Handler of Visitor

```
public interface Visitor<T, C, O, P, EL, PM, S, COA, SSA, CT>
{
  T visitVariableExp(VariableExp<C, PM> variableExp);
  T visitPropertyCallExp(PropertyCallExp<C, P> callExp);
  T visitOperationCallExp(OperationCallExp<C, O> callExp);
.
.
}
```

CompilationVisitor extends the Visitor by substituting the T parameter to Statement. As we can see in the listing above, T is associated to the return value of the handler. Considering that CompilationVisitor is developed to get the result in Statement, thus the signature of CompilationVisitor binds the generic type parameter T to specific type agument Statement.

```
public interface
     CompilationVisitor<PK, C, O, P, EL,PM, S, COA, SSA, CT, CLS, E>
```

```
extends Visitor<Statement, C, O, P, EL, PM, S, COA, SSA, CT>
```

AbstractCompilationVisitor is a type of CompilationVisitor as well as AbstractVisitor. AbstractVisitor provided by Eclipse OCL gives the flexibility to selectively override handleXxx(..) methods for internal AST nodes and visitXxx(...) methods for leaf nodes. AbstractCompilationVisitor references some of the properties of EclipseOCL i.e., Environment and EvaluationEnvironment, as it is shown in class diagram. Note that all of the components provided by Eclipse OCL are equipped with type parameters. Again, the purpose of all those type parameters is to give the generic solution to EMF and UML2 specific implementation. Environment generally stores the required variable within the compilation process of OCL expression including 'self' and another appropriate context. EvaluationEnvironment primarily developed to keep tracks of the current values of the variable within the evaluation process. For the current implementation we keep this reference just in case it is required. Furthermore this visitor references to CompilationEnvironment as well. It behaves like EvaluationEnvironment except that it functions within the compilation process instead of evaluation process.

CompilationVisitorImpl is the first concrete class of the hierarchy shown in figure 4.10. To instantiate this class, the factory class has been developed. It is encapsulated in OCLCompilation facade class which is provided to elaborate the OCL compilation functionality. The discussion regarding OCLCompilation facade class will be explained in more detail in section 4.4.3.

This concrete class implements the handler for all the leaf nodes of OCL expression metamodel. For each handler it returns the compiled expression which is represented as Statement in imperativeEcore model. Each parsed AST will be visited and compiled into Statement. Specifically, the result of visiting this OCL ASTs will return NonBlock type of Statement. When the compiled expression consists of block of statements, it returns the invocation to them, i.e., variable name which represents the block. In that case the return value is still NonBlock type of Statement. The compiled block of statements will be stored in 'result' property of CompilationVisitorImpl class which later will be combined with corresponding return value. This combination process is shifted out of the visitor implementation. To compile the inner expression of OCL ASTs, it might be recursively invoked corresponding handler operations. In this case, the caller requires only the variable which represents the representation of the compiled expressions. That is why we return only the representation variable, instead of the whole block of statements.

Listing 4.5 shows the example implementation of the visitIfExp() handler method. Mainly it retrieves all of the AST components contained in IfExp which later will be compiled as sub expression. Having all the ASTs compiled, Helper<C>, which is provided to give convenient functionality to instantiate the ImperativeEcore model, is retrieved, by invoking its getter method, getHelper(), to create the appropriate Statement result.

Listing 4.5: VisitIfExp() Handler Method implementation

```
public Statement visitIfExp(IfExp<C> i) {
  OCLExpression<C> cond = i.getCondition();
  OCLExpression<C> thenExp = i.getThenExpression();
  OCLExpression<C> elseExp = i.getElseExpression();

  Statement condStmt = (Statement) cond.accept(this);
  Statement thenStmt = (Statement) thenExp.accept(this);
  Statement elseStmt = (Statement) elseExp.accept(this);

  Block thenBlockStmt = getHelper().createBlockStmt(thenStmt);
```

```
  Block elseBlockStmt = getHelper().createBlockStmt(elseStmt);
  IfStatement ifStmt =
     getHelper().createIfStmt
           (condStmt, thenBlockStmt, elseBlockStmt);

  return ifStmt;
}
```

4.4.2 Handler Methods of CompilationVisitor

Generally the result of handler methods provided by CompilationVisitorImpl can be divided into 2 categories: Block and NonBlock. NonBlock is a representation of single statement of Java expression, while Block consists of multiple statements. Those results are the direct implication of the representation of Java expression which depends on the input AST, for instance IntegerLiteralExp and CollectionLiteralExp has different Java expression representation. IntegerLiteralExp can simply be represented by single statement in Java, as well as in ImperativeEcore. On the other hand, to represent CollectionLiteralExp, Java expression obviously requires more than one statement, i.e, first statement would be the declaration variable with the assignment to the correct kind of Collection, and the rest of the statements would be the assignment to the content of the collection. Considering that requirement, we have already categorized the representation of each OCL AST. Table 4.2 shows the type of Imperative Ecore representation for each OCL expression. Note that not all the OCL Expression constructs include in the table. For the moment, this implementation merely covers those expressions included in the table.

OCL Expression	ImperativeEcore
PrimitiveLiteralExp	NonBlock
NullLiteralExp	NonBlock
InvalidLiteralExp	NonBlock
EnumLiteralExp	NonBlock
CollectionLiteralExp	Block
IfExp	NonBlock
VariableExp	NonBlock
LetExp	Block
PropertyCallExp	NonBlock
AssociationClassCallExp	NonBlock
OperationCallExp	NonBlock
IterateExp	Block
IteratorExp	Block

Table 4.2: ImperativeEcore Representation Type of OCL Expression

Based on that, the handlers will behave differently against both result types. For the case of NonBlock, it simply returns the NonBlock representation. The example implementation for such handler has been captured in listing 4.5. For the case of Block, the handlers will create a NonBlock representation of the corresponding Block then it returns the NonBlock representation. The corresponding Block will be stored in the 'result' property of CompilationVisitorImpl. In the case of the expression has several inner expressions, this property will be accumulated by the block of statements begin with the most inner expression. In the next step, which will be explained next

in subsection 4.4.3, the caller will append the return value, NonBlock type, to the 'result' property to become one complete Block statement.

4.4.3 OCLCompilation Facade Class

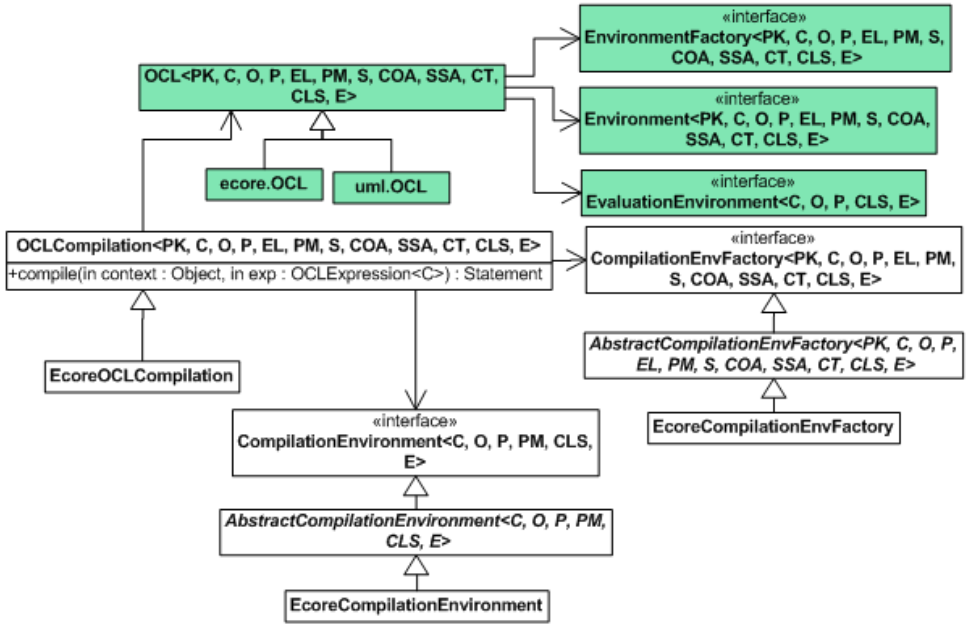

Figure 4.11: Class Diagram of OCLCompilation

This class is provided for compilation of OCL expressions. It has a reference to another facade developed by Eclipse OCL, named OCL. This reference is required for parsing the constraints. Furthermore it provides the initialization to the other references, i.e., Environment and EvaluationEnvironment. The reference to CompilationEnvFactory is essential to create the CompilationEnvironment and CompilationVisitor. The CompilationEnvironment as previously explained in section 4.4.1 is needed to provide the specific behavior in the compilation process. The detail of OCLCompilation class is illustrated in Class Diagram in figure 4.11.

Type Parameter	Ecore Type Argument
PK	EPackage
C	EClassifier
O	EOperation
P	EStructuralFeature
EL	EEnumLiteral
PM	EParameter
S	EObject
COA	CallOperationAction
SSA	SendSignalAction
CT	Constraint
CLS	EClass
E	EObject

Table 4.3: Binding of Ecore Type Parameters

The specific Ecore implementation, i.e., EcoreCompilationEnvFactory, EcoreCompilationEnvironment, and EcoreOCLCompilation, binds the provided type parameters to the corresponding type of Ecore implementation. The complete mapping of the type parameter binding listed in the table 4.3.

The main operation of OCLCompilation is shown in class diagram, named compile(). That operation performs compilation of given OCL ASTs by delegating the execution to CompilationVisitor. CompilationVisitor can be constructed while OCLCompilation has the reference to CompilationEnvFactory which has the capability to instantiate it. By visiting the ASTs of OCL expressions, CompilationVisitor results in compiled OCL Expressions represented in NonBlock. The result of CompilationVisitor, which should be a NonBlock type, will be appended with the accumulated result stored in 'result' property of CompilationVisitorImpl, if any, to form a complete Block statement of ImperativeEcore model. This final result represents the return value of the compile() method. This process is ilustrated in figure 4.12.

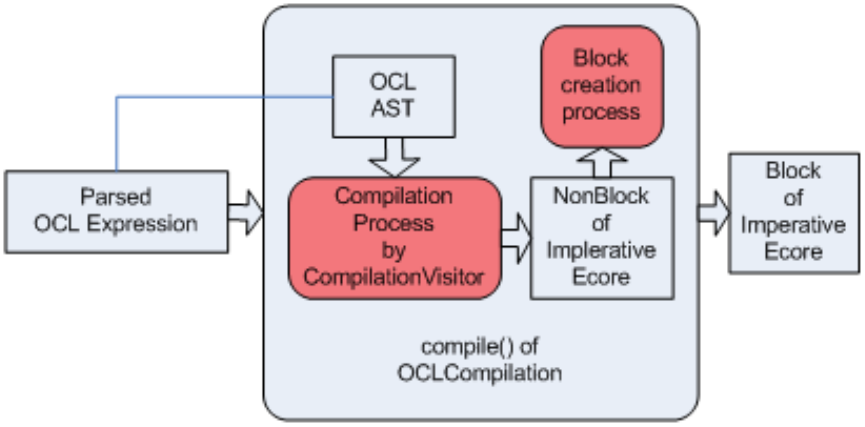

Figure 4.12: Compile process of OCLCompilation

4.5 Results and Limitation

This OCL compiler has been tested on various types of OCL expressions. In this section we show the result of the testing with the Royal and Loyal study case. Royal and Loyal example provides a large amount of OCL expressions which involves a variety of its type. The listing of complete OCL expressions can be found in appendix C.

Of the OCL expressions that we have tested, most of the expressions are successfully compiled into Java expression. The following demonstrates the example of the OCL expression and the successfully compiled expression respectively.

```
context Customer
  def:wellUsedCards:Set(CustomerCard)
          = cards->select(transactions.points->sum()>10000)
```

Listing 4.6: Example of Compiled expression

```
public EList<CustomerCard> defWellUsedCards() {
  Collection<CustomerCard> select_1 =
        CollectionUtil.createNewSet();
  for(CustomerCard i_CustomerCard : this.getCards()) {
```

```
    if((Integer)CollectionUtil.sum(
                    collect_2(i_CustomerCard))>10000){
       select_1.add(i_CustomerCard);
     }
   }
   return new BasicEList<CustomerCard>(select_1);
}

public EList<Integer> collect_2(CustomerCard i_CustomerCard){
 Collection<Integer> collect_2 =
         CollectionUtil.createNewBag();
 for(Transaction i_Transaction:i_CustomerCard.
                                    getTransactions()){
    Integer bodyExpResult = i_Transaction.getPoints();
    if (bodyExpResult != null) {
      collect_2.add(bodyExpResult);
    }
   }
   return new BasicEList<Integer>(collect_2);
}
```

However there are a number of expressions which could not be compiled to Java code correctly. This compiler has not been designed to cover all of the OCL constructs at the first place. There are some limitations that we do not implement at this stage of development. Thus we have categorized the roots of the problem of un-compiled expressions as clarified in the following subsection.

4.5.1 Unparsed OCL Expressions

Prior to generating the OCL expressions into Java code, the OCL expressions must be parsed. The parser building block has been provided by Eclipse OCL. One reason of unsuccessful compiling the expressions is the failure to pass this parsing process. When the OCL expressions can not succeed to pass on this step, the parse exception will be thrown by Eclipse OCL which causes the corresponding OCL expression can no longer be processed. The examples of Unparsed OCL expressions by Eclipse OCL are as follow:

```
context CustomerCard::getTransactions(from:Date, until:Date)
    :  Set(Transaction)
body:  transactions->select(date.isAfter(from) and
       date.isBefore(until))


context CustomerCard
   inv:  goodThru.isAfter(Date::now)
```

The first expression results in the error message of *Constraint must be boolean-valued on operation "getTransactions"*. The second expression uses the static variable which it is not supported either by EMF or Eclipse OCL.

4.5.2 Unsupported Part of OCL Expression

Figure 4.13 shows which kind of OCL expressions that is not supported by this compiler, marks with red. In fact there are no cases of Royal and Loyal involving that kind of OCL expressions.

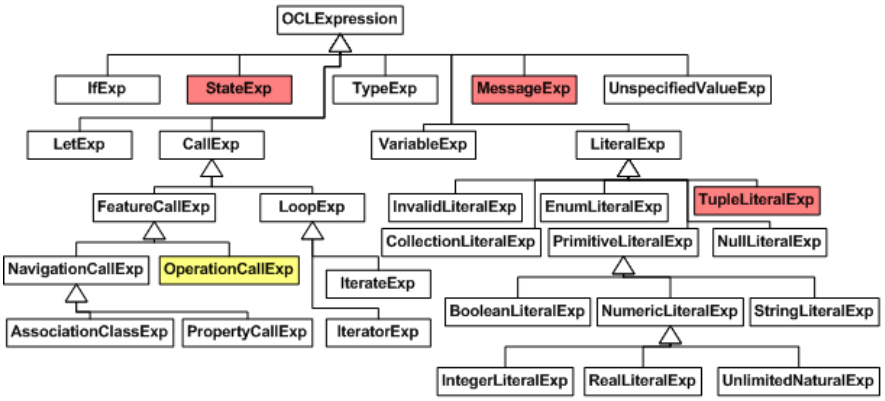

Figure 4.13: Unsupported Part of OCL Expression

Block which marks with yellow indicates that is not fully supported by the compiler which means there are small part of the expressions that has not been implemented yet. There are a lot of operations of OperationCallExp which has to be implemented in order to provide the complete supports of the expression. In this implementation some part of it has currently been omitted.

```
context LoyaltyProgram
    def:sortedAccounts:Sequence(LoyaltyAccount)=
           self.Membership.account->sortedBy(number)
```

The above expression demonstrates the invocation to sortedBy() operation which is an example of unsupported operation in this implementation.

4.5.3 Advanced Construct of Certain Type of OCL Expressions

Postcondition expressions can be used with special keywords which to some extent represents the time condition of the expression, i.e., @pre and result. The former indicates the value of corresponding element at the start of the execution time. The latter indicates the return value from the corresponding operation. Both construct are generally used in Postcondition expressions. The example of the OCL expressions which involves those constructs can be seen in the following:

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


context Transaction::program():LoyaltyProgram
    post:result=self.card.Membership.programs
```

Such advanced constructs currently are not supported by this implementation of OCL compiler. As a result, the parser can not parse the Expression in which the advanced constructs is included. The other advanced constructs are oclIsNew, isSent, and message operator.

4.6 Summary

In this chapter, the complete implementation detail of OCL compilation process has been described. ExpressionGenerator Class, the main class of this implementation,

has 4 main operation: preGenerate(), generate(), toJavaString(), postGenerate(). The responsibility to perform the tasks is delegated to Generator abstract class. Concrete class which inherits the Generator abstract class, represents the generator in which it supports the input type. GenModelGenerator class is provided in order to carry out GenModel as the valid input. The compilation process generally involves Visitor to achieve the result. The final outcome of this process is the extended Ecore model incorporated with the compiled OCL expression in its element. At the end, the limitation of the implementation is conveyed.

Chapter 5

Summary and Outlook

5.1 Summary

Aligned with the objective of this thesis, in this report we have implemented the development of basic OCL compiler which is based on EMF. The purpose of this result is to complete the missing feature of OCL implementation within EMF. Eclipse Modeling Project has developed APIs for parsing and evaluating OCL constraints and queries on EMF models. However, prior to this implementation, the compilation of OCL expression into Java expression within EMF has never been realized

At the beginning we have reviewed the strong point of EMF and its supporter who supports the OCL implementation for EMF. And there are several projects ready which worked in this OCL compiler area even though it does not support EMF. This project provides the comparable implementation except that it operates within EMF.

Furthermore, EMF has introduced the new influential feature for implementing OCL, named Generics. It makes the OCL implementation closer to the specification. Prior to that, the evaluation of OCL of, for example, element of Collection type can not be identified.

Implementing OCL compiler, we have designed the intermediate representation of Java expression, called ImperativeEcore model. It models Java expression of OCL AST in Ecore with OCL type instead of Ecore type. This design decision is taken into account in order to provide the extensible and maintainable implementation in the future.

The OCL compiler has been developed with the approach architecture described in section 3.2. The front end compiler is supported by the ready implementation of Eclipse OCL. At the end Ecore Code generator is invoked to perform the final process which generates the java expression out of OCL expression.

The evaluation of OCL expressions at runtime provides the basic yet powerful facility to EMF. Compilation OCL expression into Java expression makes the process even better. It results in advance performance while it does not require in interpreting the OCL expression at runtime. By implementing compilation OCL expression into Java expression within EMF, the benefits are:

- The Java expression can be retrieved out of OCL expression, thus the performance in evaluating OCL expression can be improved.
- Compiling OCL expression within EMF, the modeler can make use the power of EMF as well, i.e., the rich feature of generated code, editor framework, etc.

5.2 Further Work

There a few key points which require to be achieved based on this result of the OCL compilation implementation in the future:

- Currently not all OCL expression constructs has been implemented, i.e., TupleLiteralExp and MessageExp.
- At present not all OCL types has been covered in this implementation, i.e., TupleType and MessageType.
- Providing the better implementation for Init expression, while currently EMF does not support constructor code in their code generator.
- The improvement of ImperativeEcore metamodel can also be considered to attain the enhanced implementation of this OCL Compiler.
- The current implementation only provides the Ecore binding of type parameters. Providing the UML2 binding will complete the compilation process of OCL in Eclipse Modeling Project.
- Besides that, the progress of development of Eclipse OCL keeps on going. This implementation is expected to be aligned with that progress.

Appendix A

Ecore Metamodel

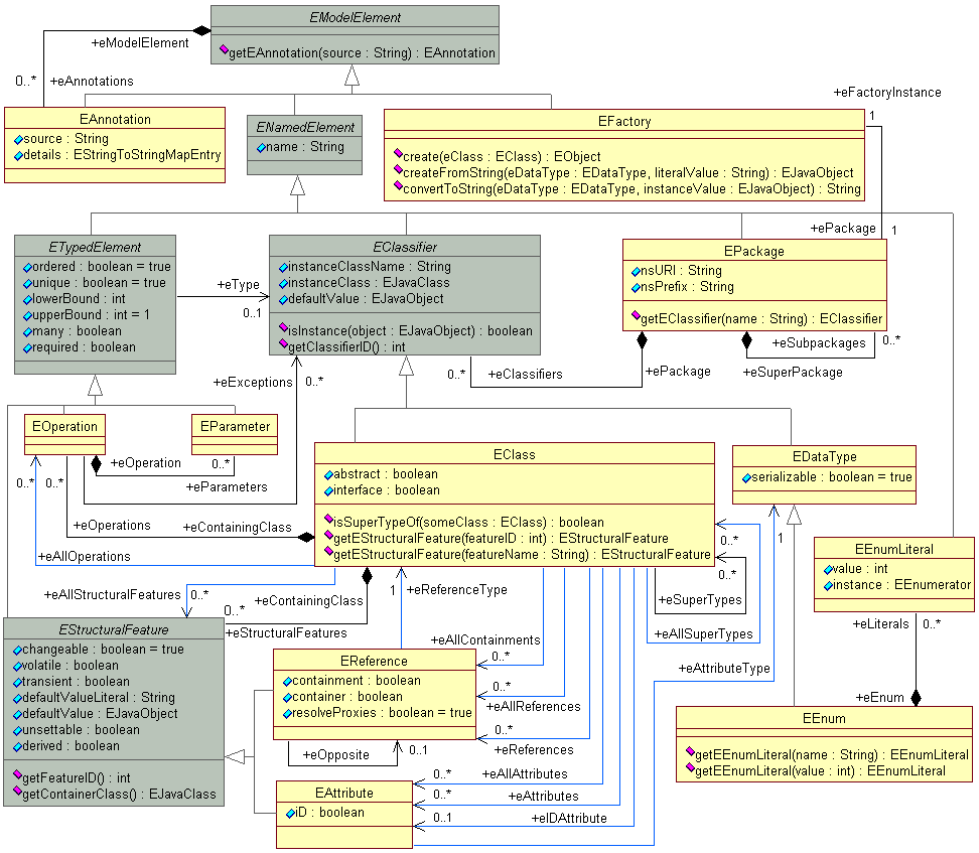

Figure A.1: Ecore Metamodel [Tea06]

Appendix B

Royal and Loyal Model

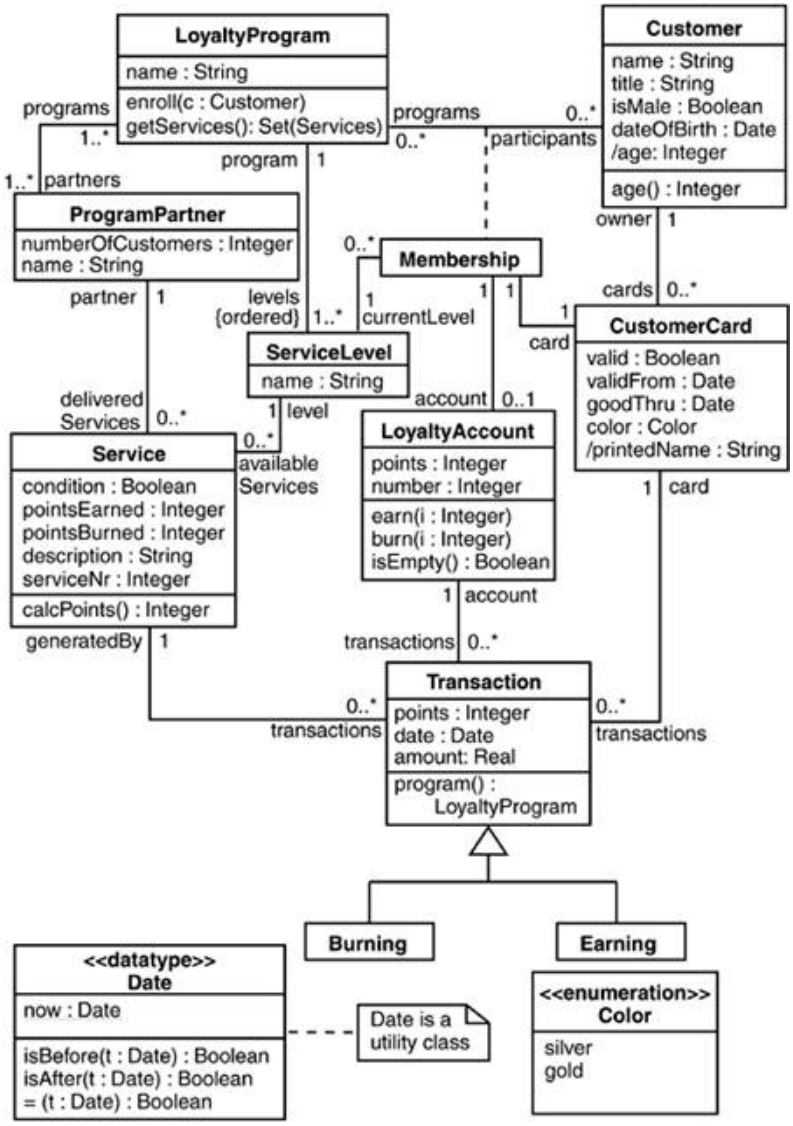

Figure B.1: Royal and Loyal Model [WK03]

Appendix C

OCL Expressions of Royal and Loyal

Customer.ocl

```
package RandL

context Customer
  inv ofAge: age >= 18

context Customer
  inv sizesAgree:
  programs->size() = cards->select( valid = true )->size()

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

context Customer
  inv: cards->select( valid = true )->size() > 1

context Customer
  inv: name = 'Edward'

context Customer
  inv: self.name = 'Edward'

context Customer
  inv: self.name = 'Edward'
  inv: self.title = 'Mr.'

context Customer
  inv: self.name = 'Edward' and self.title = 'Mr.'
```

```
context Customer
  inv myInvariant23: self.name = 'Edward'

context Customer
  def: initial : String = name.substring(1,1)

context Customer
  inv: gender = Gender::male implies title = 'Mr.'

context Customer
  inv: Membership.account->select( points > 0 )->isEmpty()

context Customer
  inv: Membership.account->reject( not (points > 0) )->isEmpty()
  inv ANY: self.Membership.account->any( number < 10000 )->isEmpty()

context Customer
  inv: self.programs->collect(partners)->
  collectNested( deliveredServices )->isEmpty()
  inv:  Set{1,2,3}->iterate( i: Integer; sum: Integer = 0 | sum + i ) = 0

context Customer
  inv: programs->size() = cards->select( valid = true )->size()

endpackage
```

CustomerCard.ocl

```
package RandL

context CustomerCard::valid
  init: true

context CustomerCard::printedName
  derive: owner.title.concat(' ').concat(owner.name)

context CustomerCard
  inv checkDates: validFrom.isBefore(goodThru)

context CustomerCard
  inv ofAge: owner.age >= 18

context CustomerCard
  inv THIS: let correctDate : Boolean =
  self.validFrom.isBefore(Date::now) and
  self.goodThru.isAfter(Date::now)
  in
  if valid then
  correctDate = false
  else
  correctDate = true
  endif
```

```
context CustomerCard
  def: getTotalPoints( d: Date ) : Integer =
  transactions->select( date.isAfter(d) ).points->sum()

context CustomerCard::myLevel : ServiceLevel
  derive: Membership.currentLevel

context CustomerCard::transactions : Set( Transaction )
  init: Set{}

context CustomerCard::valid : Boolean
  init: true

context CustomerCard::getTransactions(from : Date, until: Date )
  : Set(Transaction)
  body: transactions->select( date.isAfter( from ) and
  date.isBefore( until ) )

context CustomerCard
  inv: goodThru.isAfter( Date::now )

context CustomerCard
  inv: self.owner.dateOfBirth.isBefore( Date::now )

context CustomerCard
  inv: self.owner.programs->size() > 0

context CustomerCard
  inv: self.transactions->select( points > 100 )->notEmpty()

endpackage
```

Burning.ocl

```
package RandL

context Burning
  inv: self.oclIsKindOf(Transaction) = true
  inv: self.oclIsTypeOf(Transaction) = false
  inv: self.oclIsTypeOf(Burning) = true
  inv: self.oclIsKindOf(Burning) = true
  inv: self.oclIsTypeOf(Earning) = false
  inv: self.oclIsKindOf(Earning) = false

endpackage
```

LoyaltyAccount.ocl

```
package RandL

context LoyaltyAccount::points
```

```
  init: 0

context LoyaltyAccount::isEmpty(): Boolean
  pre : true -- none
  post: result = (points = 0)

context LoyaltyAccount::usedServices : Set(Service)
  derive: transactions.generatedBy->asSet()

context LoyaltyAccount::points : Integer
  init: 0

context LoyaltyAccount::transactions : Set(Transaction)
  init: Set{}

context LoyaltyAccount::getCustomerName() : String
  body: Membership.card.owner.name

context LoyaltyAccount
  inv oneOwner: transactions.card.owner->asSet()->size() = 1

context LoyaltyAccount::totalPointsEarned : Integer
  derive: transactions->select( oclIsTypeOf( Earning ) )
  .points->sum()

context LoyaltyAccount
  inv points: points > 0 implies transactions->exists(t | t.points > 0)

context LoyaltyAccount
  inv transactions: transactions.points->exists(p : Integer | p = 500 )

endpackage
```

LoyaltyProgram.ocl

```
package RandL

context LoyaltyProgram::getServices(): Set(Service)
  body: partners.deliveredServices->asSet()

context LoyaltyProgram::getServices(pp: ProgramPartner) : Set(Service)
  body: if partners->includes(pp)
   then pp.deliveredServices
   else Set{}
  endif

context LoyaltyProgram
  def: getServicesByLevel(levelName: String): Set(Service)
  = levels->select( name = levelName ).availableServices->asSet()

context LoyaltyProgram
  inv knownServiceLevel: levels->includesAll(Membership.currentLevel)
```

```
context LoyaltyProgram
  inv minServices: partners.deliveredServices->size() >= 1

context LoyaltyProgram
  inv noAccounts: partners.deliveredServices->forAll(
   pointsEarned = 0 and pointsBurned = 0 )
  implies Membership.account->isEmpty()

context LoyaltyProgram
  inv firstLevel: levels->first().name = 'Silver'

context LoyaltyProgram::enroll(c : Customer)
  pre : c.name <> ''
  post: participants = participants@pre->including( c )

context LoyaltyProgram::enroll(c : Customer)
  pre : c.name <> ''
  post: participants = participants@pre->including( c )

context LoyaltyProgram::addTransaction( accNr: Integer,
    pName: String,
    servId: Integer,
    amnt: Real,
    d: Date )
  post: let acc : LoyaltyAccount =
  Membership.account->select( a | a.number = accNr )->any(true),
newT : Transaction =
  partners-> select(p | p.name = pName).deliveredServices
  ->select(s | s.serviceNr = servId).transactions
  ->select( date = d and amount = amnt )->any(true),
card : CustomerCard =
  Membership->select( m | m.account.number = accNr ).card->any(true)
    in acc.points = acc.points@pre + newT.points and
   newT.oclIsNew() and
   amnt = 0 implies newT.oclIsTypeOf( Burning ) and
   amnt > 0 implies newT.oclIsTypeOf( Earning ) and
   acc.transactions - acc.transactions@pre = Set{ newT } and
   card.transactions - card.transactions@pre = Set{ newT }

context LoyaltyProgram
  def: isSaving : Boolean =
   partners.deliveredServices->forAll(pointsEarned = 0)

context LoyaltyProgram::selectPopularPartners( d: Date ): Set(ProgramPartner)
  post: let popularTrans : Set(Transaction) =
   result.deliveredServices.transactions->asSet()
   in
  popularTrans->forAll( date.isAfter(d) ) and
  popularTrans->select( amount > 500.00 )->size() > 20000

context LoyaltyProgram::enroll(c : Customer)
  pre : not participants->includes(c) -- fout
  post: participants = participants@pre->including(c)
```

```
context LoyaltyProgram::addService(p: ProgramPartner,
   l: ServiceLevel,
   s: Service)
  pre: partners->includes( p )
  pre: levels->includes( l )
  post: partners.deliveredServices->includes( s )
  post: levels.availableServices->includes( s )

context LoyaltyProgram
  inv: levels->includesAll( Membership.currentLevel )

context LoyaltyProgram
  inv: self.levels->exists(name = 'basic')

context LoyaltyProgram
  inv:  Set { 1 , 2 , 5 , 88 } ->isEmpty()
  inv:  Set { 'apple' , 'orange', 'strawberry' } ->isEmpty()
  inv:  OrderedSet { 'apple' , 'orange', 'strawberry', 'pear' } ->isEmpty()
  inv:  Sequence { 1, 3, 45, 2, 3 } ->isEmpty()
  inv:  Sequence { 'ape', 'nut' } ->isEmpty()
  inv:  Bag {1 , 3 , 4, 3, 5 } ->isEmpty()
  inv:  Sequence{ 1..(6 + 4) } ->isEmpty()
  inv:  Sequence{ 1..10 } ->isEmpty()
  inv:  Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } ->isEmpty()

context LoyaltyProgram
  inv: self.participants->size() < 10000

context LoyaltyProgram
  inv: self.Membership.account->isUnique( acc | acc.number )

context LoyaltyProgram
  inv: self.Membership.account->isUnique( acc: LoyaltyAccount
  | acc.number )

context LoyaltyProgram
  inv: self.Membership.account->isUnique( number )

context LoyaltyProgram
  def: sortedAccounts : Sequence(LoyaltyAccount) =
  self.Membership.account->sortedBy( number )

context LoyaltyProgram
  inv: participants->forAll( age() <= 70 )

context LoyaltyProgram
  inv: self.participants->forAll(c1, c2 |
  c1 <> c2 implies c1.name <> c2.name)

context LoyaltyProgram
  inv: self.participants->forAll( c1 |
  self.participants->forAll( c2 |
  c1 <> c2 implies c1.name <> c2.name ))
```

```
context LoyaltyProgram
  inv: self.Membership.account->one( number < 10000 )

context LoyaltyProgram::enroll(c : Customer)
  pre : not (participants->includes(c))
  post: participants = participants@pre->including(c)
  post: Membership->select(m : Membership | m.participants = c)->
  forAll( account->notEmpty() and
  account.points = 0 and
  account.transactions->isEmpty() )

context LoyaltyProgram::enrollAndCreateCustomer( n : String,
  d: Date ) : Customer
  pre : true -- none
  post: result.oclIsNew() and
  result.name = n and
  result.dateOfBirth = d and
  participants->includes( result )

endpackage
```

Membership.ocl

```
package RandL

context Membership
  inv correctCard: participants.cards->includes(self.card)

context Membership
  def : getCurrentLevelName() : String = currentLevel.name

context Membership
  inv levelAndColor:
  currentLevel.name = 'Silver' implies card.color = RandLColor::silver
  and
  currentLevel.name = 'Gold' implies card.color = RandLColor::gold

context Membership
  inv noEarnings: programs.partners.deliveredServices->
  forAll(pointsEarned = 0) implies account->isEmpty()

context Membership
  inv noEarnings2: programs.isSaving implies account->isEmpty()

context Membership
  inv: account.points >= 0 or account->isEmpty()

context Membership
  inv: participants.cards.Membership->includes( self )

context Membership
  inv: programs.levels->includes( currentLevel )
```

```
context Membership
  inv: account->isEmpty()

context Membership
  inv: programs.levels ->includes(currentLevel)

endpackage
```

ProgramPartner.ocl

```
package RandL

context ProgramPartner
  inv nrOfParticipants:
  numberOfCustomers = programs.participants->size()

context ProgramPartner
  inv nrOfParticipants2:
  numberOfCustomers = programs.participants->asSet()->size()

context ProgramPartner
  inv totalPoints:
  deliveredServices.transactions.points->sum() < 10000

context ProgramPartner
  inv totalPointsEarning:
  deliveredServices.transactions
  ->select( oclIsTypeOf( Earning ) ).points->sum() < 10000

  /* the following invariant states that the maximum number of points
  that may be earned by all services of a program partner is equal
  to 10,000
  */

context ProgramPartner
  inv totalPointsEarning2:
  deliveredServices.transactions -- all transactions
  ->select( oclIsTypeOf( Earning ) ) -- select earning ones
  .points->sum() -- sum all points
  < 10000 -- sum smaller than 10,000

context ProgramPartner
  inv: self.programs.partners->select(p : ProgramPartner | p <> self)->isEmpty()

context ProgramPartner
  def: getBurningTransactions(): Set(Transaction) =
  self.deliveredServices.transactions->iterate(
  t : Transaction;
  resultSet : Set(Transaction) = Set{} |
  if t.oclIsTypeOf( Burning ) then
   resultSet->including( t )
  else
   resultSet
```

```
  endif
  )

endpackage
```

Service.ocl

```
package RandL

context Service::upgradePointsEarned(amount: Integer)
  post: calcPoints() = calcPoints@pre() + amount

context Service
  inv: self.pointsEarned > 0 implies not (self.pointsBurned = 0)
  inv:   'Anneke'.size() = 6
  inv:   ('Anneke' = 'Jos') = false
  inv:   'Anneke '.concat('and Jos') = 'Anneke and Jos'
  inv:   'Anneke'.toUpper() = 'ANNEKE'
  inv:   'Anneke'.toLower() = 'anneke'
  inv:   'Anneke and Jos'.substring(12, 14) = 'Jos'

endpackage
```

ServiceLevel.ocl

```
package RandL

context ServiceLevel
  inv: program.partners ->isEmpty()
  inv: Set { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } } ->isEmpty()
  inv: Set { 1, 2, 3, 4, 5, 6 } ->isEmpty()
  inv: Bag { Set { 1, 2 }, Set { 1, 2 }, Set { 4, 5, 6 } } ->isEmpty()
  inv: Bag { 1, 1, 2, 2, 4, 5, 6 } ->isEmpty()
  inv: Sequence { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } } ->isEmpty()
  inv: Sequence { 2, 1, 2, 3, 5, 6, 4 } ->isEmpty()
  inv: Set{1,4,7,10} - Set{4,7} = Set{1,10}
  inv: OrderedSet{12,9,6,3} - Set{1,3,2} = OrderedSet{12,9,6}
  inv: Set{1,4,7,10}->symmetricDifference(Set{4,5,7}) = Set{1,5,10}
  inv: Sequence{'a','b','c','c','d','e'}->first() = 'a'
  inv: OrderedSet{'a','b','c','d'}->last() = 'd'
  inv: Sequence{'a','b','c','c','d','e'}->at( 3 ) = 'c'
  inv: Sequence{'a','b','c','c','d','e'}->indexOf( 'c' ) = 3
  inv: OrderedSet{'a','b','c','d'}->insertAt( 3, 'X' ) =
OrderedSet{'a','b','X','c','d'}
  inv: Sequence{'a','b','c','c','d','e'}->subSequence( 3, 5 ) =
Sequence{'c','c','d'}
  inv: OrderedSet{'a','b','c','d'}->subOrderedSet( 2, 3 ) =
OrderedSet{'b','c'}
  inv: Sequence{'a','b','c','c','d','e'}->append( 'X' ) =
Sequence{'a','b','c','c','d','e','X'}
  inv: Sequence{'a','b','c','c','d','e'}->prepend( 'X' ) =
Sequence{'X','a','b','c','c','d','e'}
```

```
endpackage
```

Transaction.ocl

```
package RandL

context Transaction::program() : LoyaltyProgram
  post: result = self.card.Membership.programs

context Transaction
  inv: self.oclIsKindOf(Transaction) = true
  inv: self.oclIsTypeOf(Transaction) = true
  inv: self.oclIsTypeOf(Burning) = false
  inv: self.oclIsKindOf(Burning) = false

endpackage
```

TransactionReport.ocl

```
package RandL

context TransactionReport::name : String
  derive: card.owner.name

context TransactionReport::balance : Integer
  derive: card.Membership.account.points

context TransactionReport::number : Integer
  derive: card.Membership.account.number

context TransactionReport::totalEarned : Integer
  derive: lines.transaction->select( oclIsTypeOf( Earning ) )
  .points->sum()

context TransactionReport::totalBurned : Integer
  derive: lines.transaction->select( oclIsTypeOf( Burning ) )
  .points->sum()

context TransactionReport
  inv dates: lines.date->forAll( d | d.isBefore( until ) and
  d.isAfter( from ) )

context TransactionReport
  inv cycle: card.transactions->includesAll( lines.transaction )

endpackage
```

TransactionReportLine.ocl

```
package RandL
```

```
context TransactionReportLine::partnerName : String
  derive: transaction.generatedBy.partner.name

context TransactionReportLine::serviceDesc : String
  derive: transaction.generatedBy.description

context TransactionReportLine::points : Integer
  derive: transaction.points

context TransactionReportLine::amount : Real
  derive: transaction.amount

context TransactionReportLine::date : Date
  derive: transaction.date

endpackage
```

Bibliography

[Bra04] Gilad Bracha. Generics in the java programming language, `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`, 2004.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[Dal05] Chris Daly. Emfatic language for emf development, `http://www.alphaworks.ibm.com/tech/emfatic`, 2005.

[Dam06] C. W. Damus. Implementing model integrity in emf with emft ocl, `http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html`, 2006.

[DHMS07] Christian W. Damus, Kenn Hussey, Ed Merks, and Dave Steinberg. Effective use of the eclipse modeling framework. EclipseCon, March 2007.

[ERRJ95] E.Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Jos06] Jos Warmer and Anneke Kleppe. Octopus: Ocl tool for precise uml specifications, `http://www.klasse.nl/octopus/index.html`, Dec 2006.

[LLC05] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Implementing an ocl compiler for .net. In *In Proceedings of the 3rd International Conference on .NET Technologies*, pages 121–130, Plzen, Czech Republick, May 2005.

[MP07] Ed Merks and Marcelo Paternostro. Modeling generics with ecore. EclipseCon, March 2007.

[Obj03] Object Management Group. Mda guide version 1.0.1, `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`, Jun 2003.

[Obj06a] Object Management Group. Meta object facility (mof) core specification omg available specification version 2.0, `http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf`, Jan 2006.

[Obj06b] Object Management Group. Object constraint language omg available specification version 2.0, `http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf`, May 2006.

[Obj07] Object Management Group. Unified modeling language: Superstructure, `http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf`, Feb 2007.

[Pop04] Remko Popma. Jet tutorial part 1 (introduction to jet), `http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html`, 2004.

[SDF+04] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *Java Developers Guide to Eclipse, The, 2nd Edition*. Addison-Wesley Professional, Boston, MA, USA, 2004.

[Tea06] EMF Team. Api for the ecore dialect of uml, `http://download.eclipse.org/modeling/emf/emf/javadoc/2.3.0/org/eclipse/emf/ecore/package-summary.html`, 2006.

[Tea07a] Dresden OCL Team. Dresden ocl toolkit, `http://dresden-ocl.sourceforge.net/`, 2007.

[Tea07b] Eclipse OCL Team. Model development tools (mdt), `http://www.eclipse.org/modeling/mdt/`, 2007.

[Tea07c] Eclipse OCL Team. Tutorial: Working with ocl, `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.emf.ocl.doc/tutorials/oclInterpreterTutorial.html`, 2007.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.