Runtime Invariants Checking In Plain Old Java Object

Master Thesis

Submitted By :

Sanga Lawalata
kampret77@hotmail.com
Information And Media Technologies
Matriculation Number: 29946

Supervised by :

Prof. Dr. Ralf MÖLLER
STS - TUHH

Prof. Dr. Helmut WEBERPALS
Institut für Rechnertechnologie – TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
31 October 2006

Declaration

I declare that:

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

Hamburg, 31 October 2006

Table of Contents

Abstract

Model Driven Architecture introduces the concept of Platform Independent Model (PIM), Platform Specific Model (PSM), and transformation tools that enable auto code generation for a specific platform.

Now a days many MDA tools support these features using UML and OCL as a modeling language. A software developer can model the business objects, business rules and generate the implementation codes. Especially in Plain Old Java Object (POJO), business rules are translated into invariant checking method that must be explicitly called to ensure business object consistency. Run-time invariant checking is highly needed to ensure business objects consistency during the business object life time.

In modeling level, transformation level is actually mapping PIM into PSM that holds its own constraints. In term of DSL, both PIM and PSM must achieve well formed syntax. Such well formed-ness can be tested at the IDE level. An effective and fast run time checking is thus needed to check whether each created DSL model meets such constraints.

Run time checking in POJO is presented together with advantages and disadvantages. Later the Rete algorithm is introduced to make run time checking more efficient and effective.

1 Introduction

Object oriented (OO) approach has been widely used to produce software systems that try to fulfill the business requirements. In OO analysis phase, a software developer creates the abstraction view of the business requirements (problems to solve) which results in business objects and business processes. Sometimes, inside the business objects and processes, there are constraints that must be fulfilled. These constraints are called business rules. The abstract view of the systems is known as model and the process is called modeling (these two terms are much more emphasized in Modeling Driven Architecture - MDA). Later a software developer takes the model and makes the implementation of the model based on the programming languages and the chosen tools.

Even though the Unified Modeling Language (UML) is well known as a modeling language to model the business requirements or problem domain (use case diagram), model entities (class diagram) and their activities (activity diagram, sequence diagram), it has limitation to model the business rules. For example, "A customer must have more than one customer card" can be modeled using UML with association but UML can't define the business rule that states "All customers must have valid customer cards". To respond to this need, Object Constraint Language (OCL) is defined. With OCL invariant expression, the business rule that states "All customer must have valid customer card" can be defined.

Currently, the new software development concept is emerging, known as Modeling Driven Architecture. MDA is a model based software development that tries to increase the higher level of design regardless of the implementation platforms. By defining Platform Independent Model (PIM), a software developer can keep the abstraction of model. Using transformation tools, PIM can be translated into Platform Specific Model (PSM) and later PSM can be used to generate the platform specific code. Using UML +OCL as a modeling language and supported by various modeling tools, MDA offers portability and interoperability in software development.

Many MDA tools offer support or facility from model authoring to auto code generation.

Thats it, with these MDA tools, a software developer can model the business objects (customer and customer card), business processes and also they can define the business rule (customer must have valid customer card) using OCL inside the model. Not only transforming the models into the specific platform code, checking that the models fulfill all the rules is also important, this activity is know as consistency checking.

For example Octopus (as one of MDA tools) transforms the models which hold the OCL to Java code [OCL03]. Consistency checking is performed by calling the generated OCL Java code. Eclipse Modeling Framework (EMF) performs consistency checking after a user asks for it [EMF03]. Meanwhile having the consistency checking during the run time is essential in the large system or enterprise application which involves many business object and business rules.

Meanwhile, the Rule base system offers a way to preserve the consistency during the object's life time. Defining an inconsistency as a condition, a rule based system will try to automatically find the object that match this condition and perform an action at run time. But rule based system itself is bound to the specific platform and they have their own way to express the rule.

1.1 Objective

At first time this project was aimed at invariant run time checking in EMF using the Rete algorithm. But due to time limitation, the new task is defined which is the consistency checking in POJO. The new transformation tools in Octopus will be defined.

1.2 Structure Of The Work

In the next chapter the overview of the-rule-based system will be explained and also which components construct the Rule based system together with Rete algorithm. A proprietary rule-based system and how it expresses an invariant will be explained briefly. Chapter 3 discusses OCL (especially invariant expression), how Octopus takes the UML and OCL model to generate Plain Old Java Objects (POJOs). Chapter 4 discusses the possibility to perform runtime OCL invariant checking in POJO. One approach is by defining a new Rule meta model and transformation tools and the other is by using the existing Octopus OCL meta model with additional transformation processes. The EMF and its functionalities will be presented for a future work, which enables the run time consistency checking in EMF.

The conclusions and opportunities for future work will be presented in chapter 5.

2 Rule Based System

Rule based system is a system that works continuously based on *If {condition} then {action}* statement (*rules*) and a set of data (*facts* in the *working memory*). It examines which condition inside the *if* statement can be fulfilled by the *facts* in the working memory. If rule's conditions are fulfilled, it will perform the *action* part. The working memory is "a kind of database of bits of factual knowledge about the world" [JES01] or something that holds all *facts*.

For example, there is a rule that says : "if there is a Customer object which attribute *age* is greater than *18* available in the *working memory*, print to the *console* a message". This *rule* can be expressed in iLog JRule rule syntax :

```
rule testAge {
      when {
            Customer(age>=18);
      }
      then {
            System.out.println(
                  "Customer with age>18 is in working memory");
      }
};
```

Because an *action* is based on the a *condition* which is fulfilled by a *fact* in the *working memory*, this approach is known as the *forward chaining* method or *data-driven* method. Below is another rule example that takes 2 *conditions* [Listing 1]. Rule *testAge2 action* part will be run (*fired*) when the system finds a *Customer* object whose *age >18* and a *CustomerCard* object which is *valid* (or *valid* equals *true*).

```
rule testAge2 {
      when {
            Customer(age>18);
            CustomerCard(valid==true);
      }
      then {
            System.out.println("testAge2 rule is fired");
      }
};
```

Listing 1: testAge2 in JRules rule

2.1 *Inefficient solution*

A rule-based system keeps a list of the *rules condition* (known as Left Hand Side/LHS) and finds all *facts* that fulfill the *LHS* (this approach is called *rules finding fact*). Considering that the *facts* can change, it means the system must recheck all the *conditions* in the *LHS* if these new *facts* can fulfill one or more *conditions* in *LHS* which results in running *RHS*. Considering *testAge2* rule example, it takes time proportional to the product $Nb_{Customer}.\ Nb_{CustomerCard}$ on each cycle where $nb_{Customer}$ is the number of *Customer* objects and $Nb_{CustomerCard}$ is the number of *CustomerCard* objects. We could write the result RF^P where R is a number of Rules, F if total number of Facts, and P is the average number of conditions per rule " [JES01].

2.2 *Rete Algorithm*

Rules finding facts approach is inefficient because the rule-based system has more or less fixed set of rules even though *facts* inside the working memory change continuously. "And also it is an empirical fact that in most rule-based system, rules inside the working memory are also fairly fixed overtime. Although new *facts* arrive and *old facts* are removed as the system runs, the percentage of facts that are changed per unit time is generally fairly small." [JES01].

The Rete algorithm, invented by Dr. Charles Forgy [WFL01], is divided into two parts which are rule compilation and runtime execution (which will be explained later). Compared to the *rule finding facts* approach, Rete algorithm breaks down rules into a series of trees where each nodes has memory to store *facts* so that if some *facts* are changed, the rule-based system just needs to reevaluate the *rule* whose condition is affected by the changes and it doesn't reevaluate all the *rules*. Rete algorithm also has mechanism to test the *fact* that is related with specific *rules* and not testing all *facts* with all *rules*.

2.2.1 Preliminary Rete Algorithm

2.2.1.1 Data Type

At the earlier stage implementation, Rete was implemented without known object oriented concept. A fact of "a customer whose *age* is *18"* can be defined by *(18)*. And another *fact john* as customer's name and the two will be *(18 john)*. This *fact* is an *ordered fact* because

(18 john) is different with *(john 18)*. A *fact structure* is presented in order to represent the *facts* regardless of their position. A fact structure consists of *fact slot* and *fact template*. A *fact slot* defines a value or a *fact* such as *18* or *john*. Meanwhile a *fact template* is a groups of *fact slots*. Below are two examples how to define the *template* and *fact template* to represent unordered *facts*.

```
(Customer (name john) (age 18)  (male true) )
```

Listing 2: To define a fact using a template

```
(deftemplate Customer
        (slot name)
        (slot age)
        (slot male))
```

Listing 3: The Fact Template Structure

At that time, the rule language itself didn't define the data type. *18* (in *age*), *john* (in *name*), *true* (is *male*) don't have a type of *integer*, *string*, or *boolean*. Even though the rule language doesn't define the data type, but the underneath language in the rule engine has its own data type [MXE01] . This is how to express *testAge2* using *fact template* as a *condition*:

```
defrule testAge2(
AND(
            customer (name ?name) (age 18) (male ?male)
            customerCard (valid true)
))
=> "do some action"
```

Listing 4: testAge2 rule

```
defrule testAgeb(
AND(
            customer (name ?name) (age 20) (male ?male)
            customerCard (valid true)
))
=> "do some action"
```

Listing 5: testAge2b rule

2.2.1.2 Rule Compilation

In *Rule* Compilation, a *rule* is translated into Rete tree network which consists of *RootNode*, *OneInputNode* or *AlphaNode*, *TwoInputNode* or *BetaNode*, and *TerminalNode*. All *facts* enter the Rete tree network through a RootNode. From RootNode, every *fact* propagates into the *AlphaNode*. A *BetaNode* receives two inputs from 2 *AlphaNodes*. A *fact* will be filtered inside the *AlphaNode* based on the *constrains*. The *constrains* , in this case is a *fact* or an *open variable*. *AlphaNode* compares the incoming *fact* if it matches *AlphaNode constraints*. Each *AlphaNode* remembers all *facts* that match its *constraint* and pases the *facts* into the next node. A *BetaNode* takes input from 2 *AlphaNodes* and does operations such as doing comparation or *AND* operation, and passes all the *facts* down to *TerminalNode*. Let's take *testAge2* as an example and below is the Rete tree for *testAge2*.

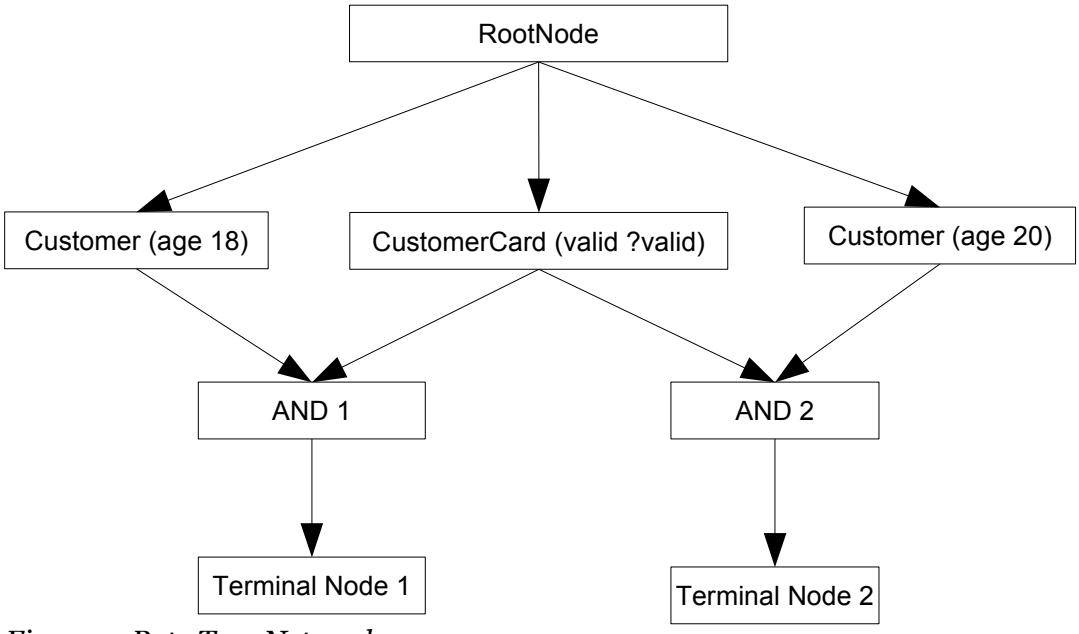

Figure 1: Rete Tree Network 1

```
(Customer (name john)  (age 18)  (male true) )
(CustomerCard (valid true))
(Customer (name marry) (age 20)  (male false) )
(CustomerCard  (valid false))
```

Listing 6: Facts presented in fact template

When *a fact* of *john* (defined with *a Customer fact template*) enters the *RootNode*, it propagates to *AlphaNode Customer(age 18)*, *CustomerCard(valid ?valid),* and *Customer (age 20)*. *AlphaNode Customer(age 18)* will compares the *fact*, first with its *fact template* name (*Customer*), *slot name* (*age*), and s*lot value* (*18*).If they are equal, *AlphaNode Customer (age 18)* saves this fact inside its *AlphaMemory* and sends the fact down to the *BetaNode And1*. *AlphaNode Cutomer (age 18)* ignores the fact *CustomerCard (valid true)*. The *AlphaNode CustomerCard (valid ?valid)* takes all *CustomerCard fact*. *?valid* is an *open variable* which stores *value* to be used later. *AlphaNode CustomerCard(valid ?valid)* stores *CustomerCard(valid true)* and *CustomerCard(valid false)* in its *AlphaMemory* and sends them to the *BetaNode AND1* and *AND2*.

BetaNode AND1 receives inputs *Customer(name john) fact* which is stored in *AND1 LeftMemory* , and *CustomerCard(valid true)*, *CustomerCard(valid false) facts* which are stored in *AND1 RightMemory*. Later, *AND1* will send all the *facts* from *AND1 LeftMemory* and *RightMemory* into *TerminalNode*. *TerminalNode* calls the *action* has been defined and takes all *facts* from the above nodes.

In the Rete Network *testOfAge2* [Figure 1], the node *CustomerCard(valid? valid)* sends output into 2 *BetaNodes*. It is called s*haring node* which is one of the Rete optimization. It will be explained in more details in the *ReteOO*.

2.2.2 Rete OO

2.2.2.1 Data Type

In the Rete OO implementation, instead of using *fact template* and *fact slot*, it adopts Object Oriented data type which defines *classifier*, *attributes*, and *value*. In short *fact template* and *fact slot* are similar with *classifier* and its *attributes*. Adopting this new data type, new types of *AlphaNode* are introduced, such as *ObjectTypeNode*, *LiteralConstrainstNode*, etc.

2.2.2.2 Rule Compilation

In Rule Compilation, a rule is translated into Rete Tree networks which consists of *RootNode*, *AlphaNode* , *BetaNode* and *TerminalNode*. All objects enter the network through the RootNode. From there, they will go to the *ObjectTypeNode*. *ObjectTypeNode* will filter the object based on its object type (or *class*/*classifier*). The system doesn't have to evaluate every single node for every single object by attaching one or two input nodes after *ObjectTypeNode*. When a new *Customer* object is inserted into working memory, it will not go or propagate into nodes that aren't for *Customer* object.

AlphaNode is used to evaluate *literal conditions*. *age>18* is a Customer *literal condition* or *literal constraint*. If a rule has more than one *literal condition* for one object, there are a few *AlphaNodes* that are linked to each other. One object must satisfy the first *literal constraint* before it can proceed to the next *AlphaNode*. *ObjectTypeNode* which has been mentioned before can be considered as a specialized *AlphaNode* [WFL01].

BetaNode is used to compare 2 objects which either a different or the same object type. It receives the list or object for its right input and receives one object for its left input.

At the end of tree node for a rule is *TerminalNode*. It indicates that a *rule's* conditions have been satisfied. It will put all the *facts* and the *actions* in the *agenda* to be executed later.

2.2.2.3 Rule Runtime

In *Runtime execution*, when an object enters into working memory, the rule-based system will pass the object into the root node. The object will enter the *ObjectTypeNode* and propagates down to the network. *AlphaNodes* will put in its memory all objects that matche its condition. In the case of *testAge2* [Figure 2], *Customer's ObjectTypeNode* (blue node) will store all objects with class type *Customer*. The *AlphaNode* (red) will store all objects which match *age > 18* and *valid == true*. The *BetaNode* will store the list of *Customer* objects (taken as left input and saved in its *LeftMemory*) and a single *CustomerCard* object as a right input (saved in its *RightMemory*). If all conditions are satisfied (in this case *AND),* this *BetaNode* sends all the objects to the *TerminalNode*. *TerminalNode* will add all the *facts* (all the objects) and the *action* into *agenda* to be executed.

Because each *AlphaNode* and *BetaNode* has memory which stores all the objects, if there is a change in an object value, e.g. an object *Customer* changes its attribute *age* into *10,* then all the nodes that store this object will update their memory and reevaluate the rule tree in where there nodes are attached to.

2.2.2.4 Rete Optimization

Until now, many improvements have been done in Rete implementation. One of the improvements is node sharing. For example, there are two rules : *testAge2* and *customerAndTransaction.* They share the same condition which is *age>18*. Instead of building 2 trees that reflect the 2 rules, *AlphaNode (age>18)* can be shared between the rule.

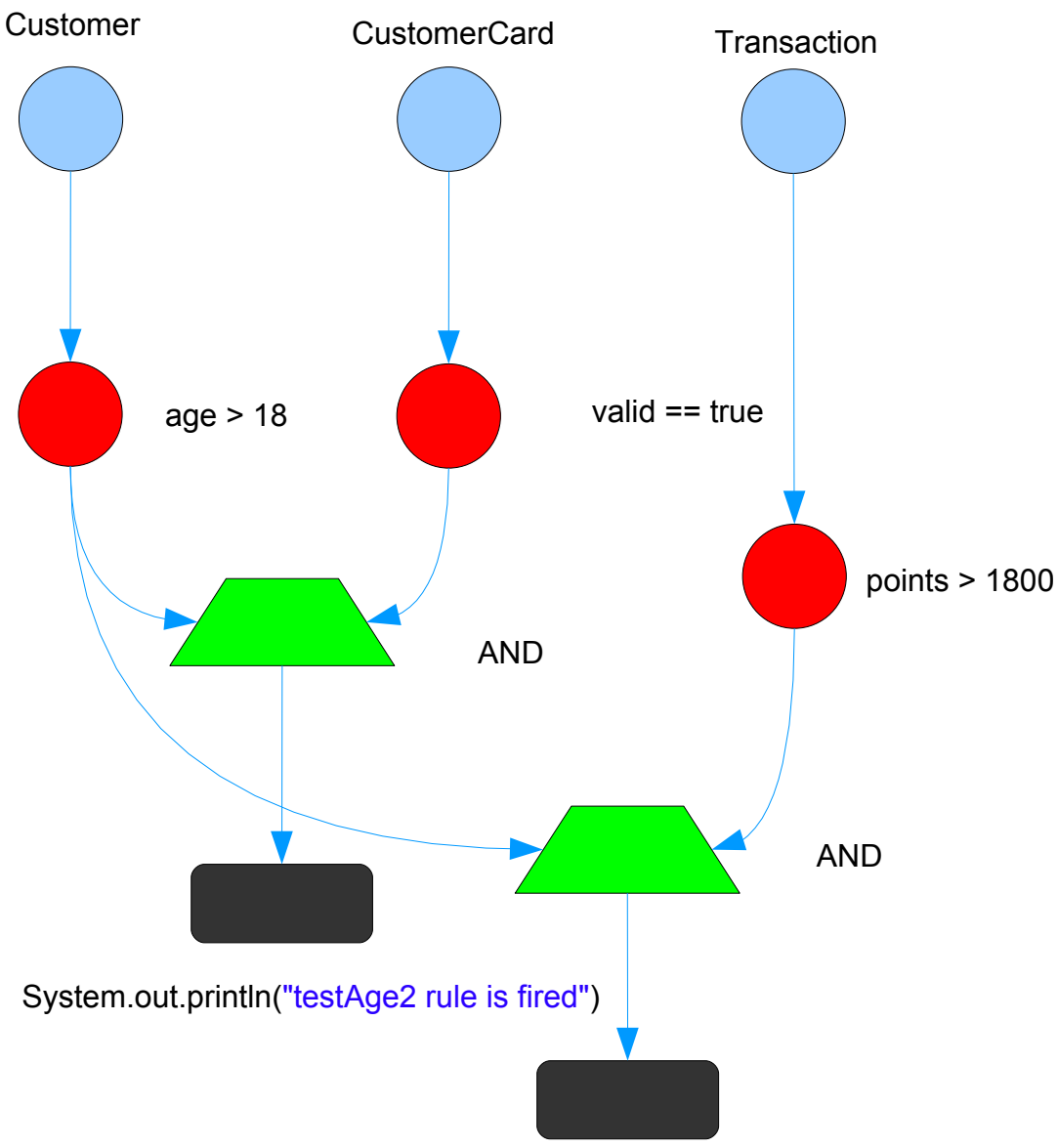

Figure 2: Rete Tree Network with Node Sharing

Node sharing optimization reduces the amount of memory used in each node because there is just one *AlphaNode age>18* instead of two *AlphaNode age>18*s, which store the same information (one *AlphaMemory*). If an an object of *Customer* changes its *age* attribute, *AlphaNode age>18* must check in its memory if the object of *Customer* still satisfies the *age>18* constraint. It means that the rule-based system needs to update only one *AlphaNode age>18* memory and automatically reevaluate all rule trees where this *Alpha Node age>18* is attached to.

```
rule customerAndTransaction {
      when {
            Customer(age>18);
            Transaction(points>1800);
      }
      then {
            System.out.println("customerAndTransaction rule is
fired");
      }
};
```

Listing 7: customerAndTransaction rule in JRules

2.3 Why Use Rule Based System

If there are a *Customer* with *age>18* and *CustomerCard* with *valid==true* as conditions and someone wants to change the condition from *age>18* into *age<17* then he/she must change the code and compile the application. In the enterprise application, if someone wants to change the conditions because of some circumstances (usually because of additional requirements or changes of business activities), changing the code and recompiling the application cost additional resource and are inefficient. Using a rule-based system, someone can just define a new rule or pull out the old rule, put it in the *working memory* without adding or compiling the running application which is much more complex than a single Java class. *Rule based* system is suitable for a system whose *facts* or conditions (or *rules*) are changing all the time or dynamically.

The separation between rule (or business logic) and application logic is one of the reasons why a software architecture considers rule based system. Look at the Java code snapshot above [Listing 20]. Someone must know in Java programming language how to specify *if* statement. Also, both objects *Customer* and *CustomerCard* are presented as variable *cus* and *cc*. To get their attributes, their *setters* must be called. Compared to the *testAge2* (written in iLog), someone can easily define that *testAge2* rule which needs a *Customer* and *CustomerCard* object. A rule can be defined regardless of the programming language that builds up the rule-based system. Using the specific characteristic of the programming language, the Rete algorithm can be optimized to speed up the matching procedures to action activation.

Separation between rule and application algorithm enables rule developer to concentrate on developing rules without necessarily knowing the application logic or the platform specific language. At this point, *rules* are considered as other assets for a company.

2.4 When NOT To Use Rule Based System

A rule-based system can't answer all problems. It is just a small solution of software engineering. "Rule Engine (or rule-based System) is not really intended to handle work flow or process executions" [DRL01]. Rule-based system can't guarantee that actions are processed in sequences, because *action* part or *RHS* depends on *LHS* and *LHS* depends on the *facts* (which match *LHS* condition) that are available in the *working memory* at that time. *testAge2* conditions doesn't emphasize that a *Customer* object must be available earlier in the working memory than *CustomerCard*. It can be that an object of *CustomerCard* is already on the working memory first and later there is an object of *Customer*. *testAge2* just states if there are *Customer (with age>18)* and *CustomerCard (valid==true)* objects in the working memory, do some *actions* (prints message to the console).

2.5 Part of Rule Based System

Based on its characteristics, usually there are several parts which *build* a rule-based system, which are :

- *Rules Management module*, accesses the repository and loads the correct *rules-set* into engine
- *The agenda* : tracks the prioritized *rules* selected by the *inference engine* during the pattern-matching logic cycle.
- *The working memory* : contains the current state of *facts* that led to the current *rules* in the *agenda*.
- *Execution Context Module* : represent the runtime environment for the *inference engine's execution*. During the *inference engine's execution* cycle (logic cycle), an execution context would hold a physical grouping between a specific instance of the *agenda* and the *working memory*. More than one execution context can simultaneously exist and share the same *rules-set*.

- *The Inference Engine* : will pass the data inserted in the *working memory* to the *root node*. From there, it enters the *ObjectTypeNode* and propagates downs the network. Later it puts *rules* into the *agenda* based on the *facts* in the *working memory*.
  If a *rule* is executed, then more *facts* are added and inserted into the *working memory* to be used later to match *rules* until the logic cycle's end, when no more *rules* can be matched with *facts* from the *working memory*.

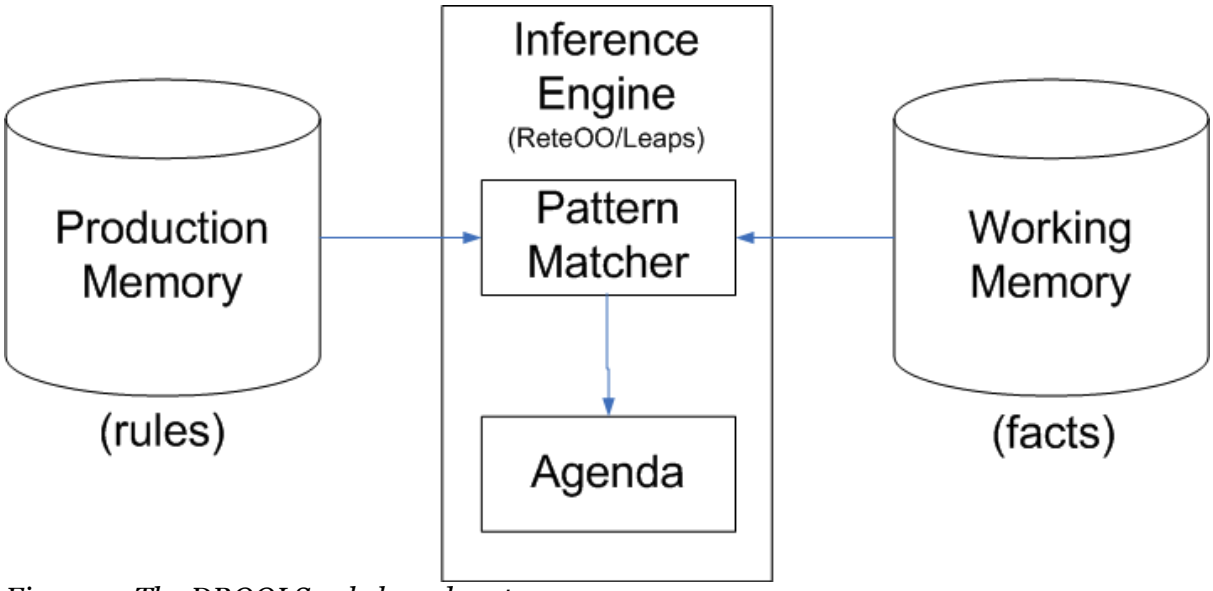

Figure 3: The DROOLS rule based system

2.6 ILOG Business Rule Studio

JRule is one of the rule-based systems which implement Rete algorithm on their rule engine and use Java as their programming language. Instead of just offering the rule based system, it offers ILOG Business Studio for business rule authoring, testing and debugging in eclipse environment.

The interaction between ILOG Business Rule Studio and JRules rule engine is the following:

- JRules rule engine executes the business rules created by business rule studio based on the available business data.
- The Application logic is not aware of the business logic. It represents the business data to the user and change the business data. Meanwhile the JRules rule engines can interact with the application logic.

- To create, test, and debug a *rule*, a rule developer uses the ILOG Business Rule Studio. When the rules are ready, they will be deployed in the JRules rule engines. Rules can be added when the rule engine is running. In the time a new rule is added into rule engines, the new business rule will be applied automatically.

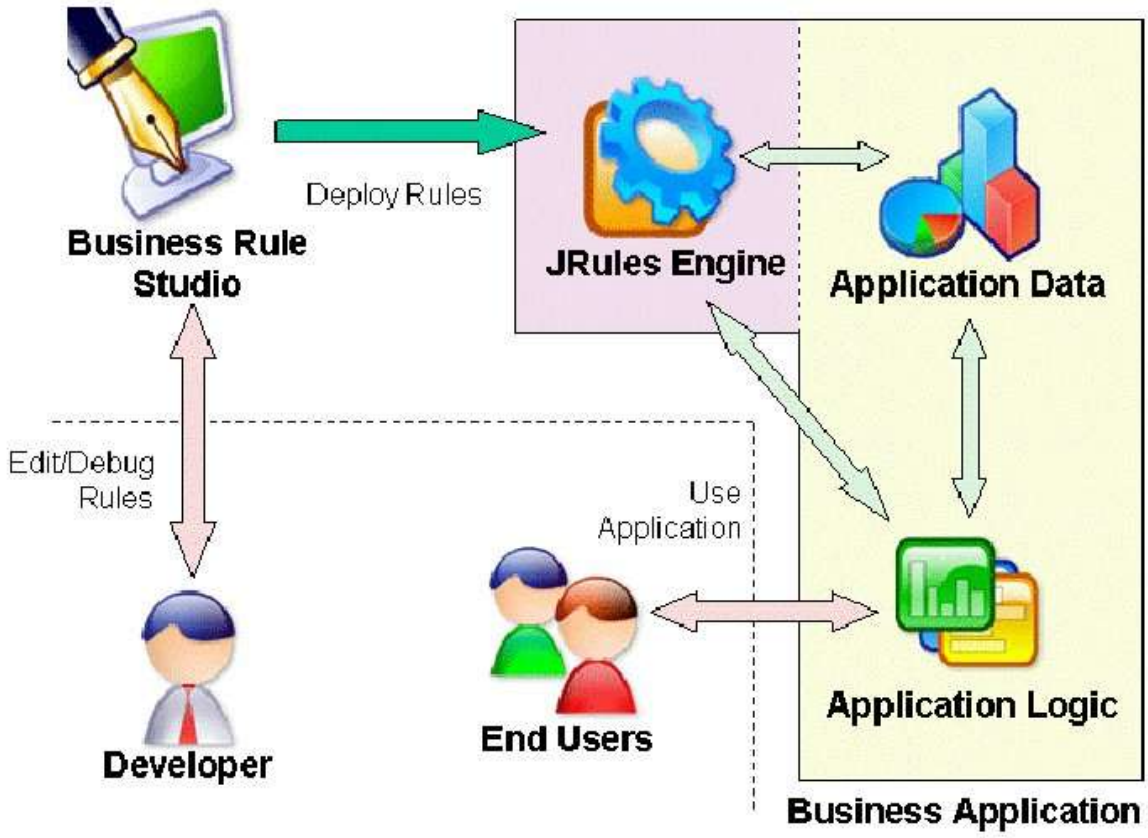

Figure 4: Scenario between ILog Business Studio, JRule rule engine, and Application

For these purposes, ILOG Business Rule Studio offers 2 perspective layouts: *rule authoring* and *rule debugging*. *Rule authoring* layout helps rule developers to create a *rule* based on the *eXecution Object Model* (XOM). Like Java programming perspective, it provides a *rule* syntax checking, available objects, and auto completion. *Rule debugging* layout helps a *rule* developer to debug which objects (or *facts*) are available in *working memory*, which *rules* are fired, and the sequence of *rule* executions.

JRule rule engine consists of several parts which are inference engine/rule engine itself, the rule, the working memory, agenda.

2.6.1 The Rule Engine

As a system that implements Rete algorithm and is built on Java, it has two main parts which are actually a Java class. *IlrRuleset* is taking care of *rule* compilations and a Java class *IlrContext* is the *rule* engine. It has a *working memory*, *agenda*, *inference engine*. But all of them are transparent from users and rule developers.

```
// load the rule from stream
IlrRuleset ruleset= new IlrRuleset();
boolean parsed = ruleset.parseStream(stream);

// check if it is a correct set of a rules
if(!parsed) return;

// load the rule
IlrContext context =  new IlrContext(ruleset);

// insert the rule
context.insert(obj1);

//run all the rule
context.execute();
```

Listing 8: How to use JRule rule engines in Java application (load rules, insert a fact, and execute rules)

Rules are defined in a file (with .dl extension), read as a stream of text. A instance of *IlrRuleset* will take the stream as the input and transform the text into a set of Rete tree where each tree is represented by java class *IlrRule*.

A user inserts the object (or facts) through an instance of *IlrContext,* which puts the object to the *working memory*. After *execute()*, *IlrContext* performs pattern matching and puts a set of *actions* and *facts* into *agenda* to be executed.

2.6.2 Rule

Rules can be used to express the *business rules*. Considering *business rules* as conditions that must be fulfilled first (e.g : *Customer* with *age>18*) before doing some actions (e.g : allow transferring money), the *business rule* developers define all the conditions within a *rule* using ILOG Rule Language which has a number of keywords and reserved words.

```
rule ruleName {
priority  = propertyValue;
property  propertyName =  value1;
      when { conditions }
      then {  actions }
      else {  actions }
};
```

Listing 9: JRule rule structure

An IRL rule has structures as follows :

- *Header part*, defines the rule name, the priority of the rule, and properties associated with the rule.
- *Conditions part* (or *LHS*), defines conditions or a set of patterns that refers to the Java object.

  In our example, *Classifier* constraint (*Customer*) and *Integer* constraint (*age>18*) are the conditions.
- *Actions part* (or *RHS*), defines the actions if conditions are either fulfilled (in then part) or not fulfilled (in else part).

testAge JRule rule below is the example how to define business rule which is a *Customer* with *age > 18* as a *condition*.

```
rule testAge {
priority = 1;
      when {
            Customer(age>18);
      }
      then {
            System.out.println(
                  "Customer with age>18 is in working memory");
            // or transferring money
      }
};
```

2.7 Summary

In this chapter, the idea behinds Rete rule-based system is presented. Rete algorithm is used to speeds up the pattern matching based on the idea that just small *rules* are changed during the system life time. *Rules* will be translated into Rete *trees* which consist of nodes that have *Alpha* and *Beta memories*. These *memories* will remember all related facts and prevent not all *rules* be reevaluate when there is some *fact* changes.

Rete algorithm has disadvantages. It consumes more memory resources because each *node* has its own memory which stores the related facts. Node sharing is a way to reduce memory size by sharing the same node between Rete *trees*.

3 Object Constraint Language

From object oriented analysis, the conceptual model (or subsequently just called "model") of the system is defined. To define a model, a well formed language is needed so that this modeling language can be interpreted by a computer. It must be well-defined in form (syntax) and meaning (semantic).It is why a computer can't process the natural language. Today UML is widely used as an modeling language to define a model of a system. Using UML, an activity diagram can be used to model the dynamic part of the business requirements (such as business activities) and class diagram can be used to define a static part of the business requirement (business object).

But UML has limitations in expressing some business rules that are applied inside the model. For example, UML can express "a customer must have more than one customer card" using an association, but it can't express "a customer must have age>18" or "customer must have valid customer card". UML can't define the *preconditions* before executing and *postconditions* after executing an action either. In respond to these needs, Object Constraints Language is defined. Together, UML and OCL make a model more robust.

3.1 Invariant

An Invariant defines the constraint inside the model that must be valid (or true) during the constructor and completion of every public operations, but it is not necessary for it to hold during the execution of operations [OCL01]. An invariant is a boolean statement that must return true. If it returns false, it means the invariant is broken. This is how to express the business rule "all customer must have valid customer card" in OCL using invariant statement.

```
context Customer
     inv allValidCards : cards->forAll(valid=true)
```

Listing 10: allValidCard invariant

Starting with *context*, it shows in which class this invariant is applied. *inv* is the OCL keyword for invariant, *cards->forAll(valid=true)* means that cards is a *collection* and each card must have an *valid* attribute which is equal to true. Figure 5 shows the association between *Customer* and *CustomerCard* classes.

Figure 5: Association between Customer and CustomerCard classes (RandL)

3.2 Octopus

OCL Tool for Precise UML Specifications (or Octopus) is one of the modeling tools to model the business objects and business rules using UML and to add OCL expression inside your business models. Octopus is an eclipse plug-in, providing *environment* model authoring using UML and OCL. It has not only syntax checking for UML and OCL languages but also capabilities to generate the Java codes for the defined models.

Adopting Model Driven Architecture, Octopus has defined its own model of UML and OCL expression. The models that defines UML and OCL expression are called meta-model that means a model that defines another model. Octopus instantiated *IClassifier* (UML meta-model) to express the term *class* in UML meanwhile a term *class* models a business object called Customer.

These meta-models (UML and OCL meta-models) are independent from any platform. But still these meta-model are expressed using Java language. *Independent* refers to the term *Classifier*, *Attribute*, *Reference* (in UML), or *Invariant* (in OCL) not to the programming language in which these models are written. These models (UML and OCL models) are called Platform Independent Model (PIM). Octopus also has a Platform Specific Model to a targeted platform which is Java. When generating a Java code from UML and OCL models, the code generator takes the UML and OCL models as an input and creates the Java model and later the Java code is generated from the Java model. The UML and OCL models are created during the model authoring in eclipse.

Octopus consists of 3 main Java packages which are *nl.klasse.octopus* (PIM of OCL and UML model), *nl.klasse.octopus.javametamodel* (PSM of Java Model) , and *nl.klasse.octopus.codegen* (for Java code generation).

As an open source MDA tools, Octopus can be extended easily. A software developer can define other PSMs. For an instance, the project "Translation of OCL Invariants into SQL:99 Integrity Constraints". This project defines the new SQL PSM and transformation tools which transform from OCL PIM to SQL PSM [OCL01].

3.2.1 Platform Specific Model

Octopus has Java Platform Specific Model. Java PSM is needed to map UML and OCL models into Java code, which has platform specific technical details . Let's take the *Customer* in *RoyalAndLoyal* (*RandL*) *class diagram* as an example. It has an *age* as its *attribute* and it has a *public visibility*. In Java rather than setting the *age* visibility as *public* , it is better to set *age* visibility as *private* and create the *age* setter visibility as *public* (*public setAge()*). The PSM defines platform-specific technical details such as Class in Java and table in database so that a bridge can be defined between 2 PSMs. For each class model in Java PSM, a bridge creates the table model in DBMS PSM. In code level, a *Customer* Java class, a code bridge will creates the Customer table in DBMS.

Even though it sounds promising, the differences of the target platform specifications can't make all of the features mapped from one PSM into another PSM. A good PSM and platform specific technical details in this case are very essential.

3.2.2 Generated Java code for association

UML defines an *association* between classes. In some case, an *association* is equal to programming term a *pointer*. A *pointer* can be used to refer to another element. An element can be either a single object or an *n*-object collection. But a referred element usually isn't aware that another element holds a reference to it, meanwhile an *association* shows awareness of 2 *classes*. *Two way association* means both classes hold the other sides references. *One way association* means just one class holds the other class's references and it is not vice versa.

Figure 5 shows *two way association* between *Customer* and *CustomerCard* classes in generated Plain Old Java Object (POJO). *Customer* class has attribute *cards* which holds references to *CustomerCard* objects, while *CustomerCard* class has attributes *owner* which holds *Customer's* reference.

To express *awareness*, Octopus adds some additional methods to implement *association* in UML models which are :

attribute	set_()
L to 1	set_ () z_internalRemoveFrom_() z_internalAddTo_()
L to N	set <Role>(Collection) z_internalRemoveFrom_() z_internalAddTo_() addTo_(collection) addTo_() removeFrom_(collection) removeFrom_() removeAllFrom()

Listing 11: Octopus Additional Methods

3.2.2.1 LtoN Association

Customer to *CustomerCard* is an example of *LtoN association*. For example there is an object *cus1:Customer* wants to add *cc1:CustomerCard* into its list of *cards* (*f_cards*). *cus1:Customer* will do the following steps :

1. *cus1:Customer* will check if another object refers to *cc1:CustomerCard.*
2. if there is an object that refers to cc1:CustomerCard e.g cus2:Customer , *cc1:CustomerCard* cuts its reference with *cus2:Customer* by calling *z_internallyRemoveFromCards(cc1)*.
3. *cus1:Customer* sets reference to *cc1:CustomerCard.*
4. *cc1:CustomerCard* sets reference to *cus1:Customer* as its new *owner.*

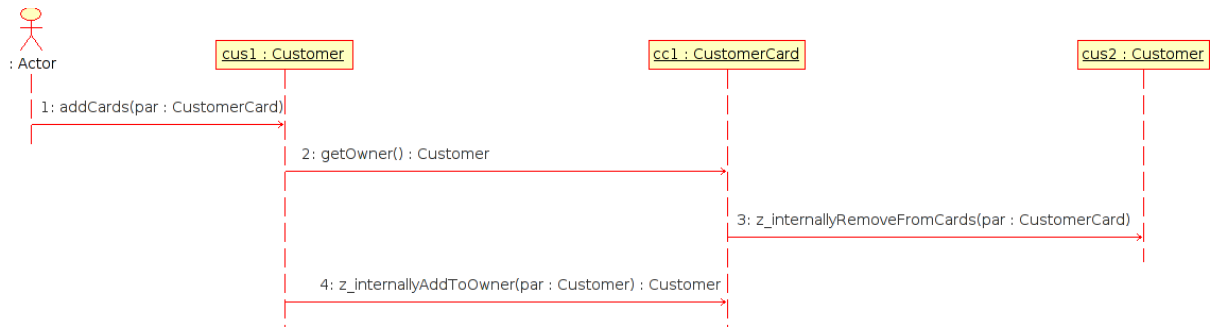

Figure 6: LtoN Sequence Diagram

3.2.2.2 LtoOne Association

CustomerCard to *Customer* is an example of *LtoOne association*. For example there is an object *cc1:CustomerCard* wants to add *cus1:Customer* as its new *owner*. *cc1:CustomerCard* will do the following steps :

1. If there is another object of *Customer*, e.g *cus2:Customer* refers to *cc1:CustomerCard* then *cus2:Customer* cuts its reference to *cc1:CustomerCard* by calling z_internallyRemoveFromCards(cc1).
2. *cc1:CustomerCard* sets reference into *cus1:Customer* as its new *owner*.
3. *cus1:Customer* will set reference into *cc1:CustomerCard* by calling *z_internallyAddToCard(cc1)*.

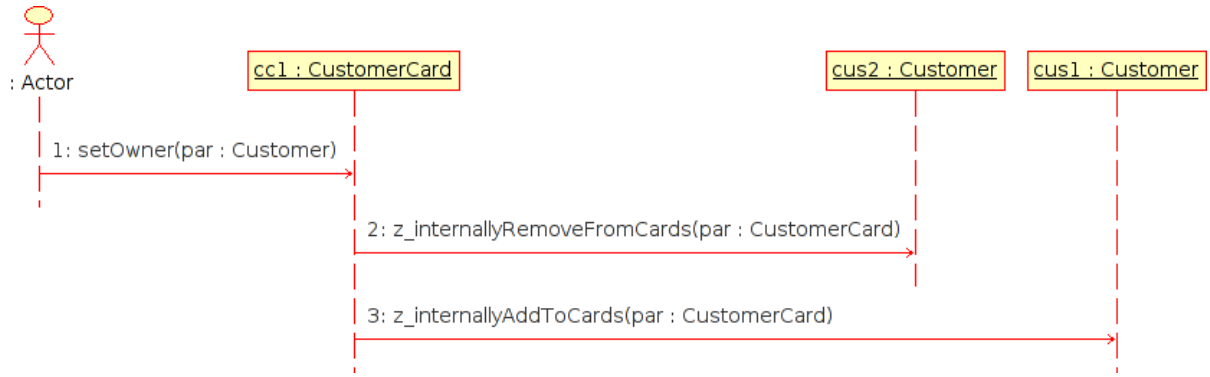

Figure 7: LtoOne Sequence Diagram

3.2.3 Octopus Code generator

The Octopus core does the following step :

1. It reads the UML files that define the UML models, parses and analyzes them if they contain errors. If they are error free, it creates the UML models by instantiating UML model classes and continues with step 2.
2. It reads the OCL files, parses and analyzes them against the UML models. If they are all correct, the core adds the OCL expression in the UML model by instantiating OCL Expression model classes and putting them inside the UML model (inside *ClassImpl* object) based on OCL expression context. It there is an error, it shows an error message.
3. If a user changes the UML model, it goes to step 1 and if a user changes the OCL files, it goes to step 2.

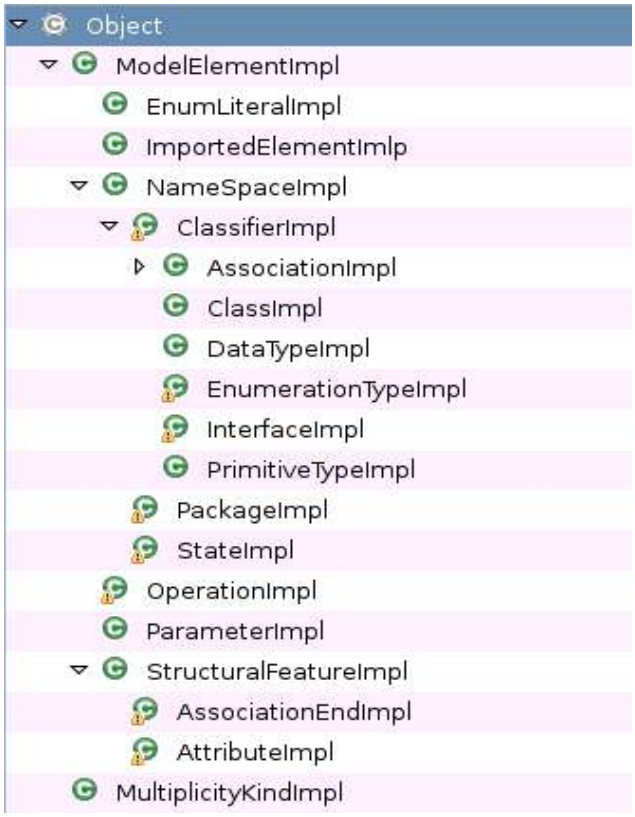

Figure 8: UML Meta-model Structure

3.2.4 Visitor Pattern

To generate the Java code, first the code generator takes the UML model as an input and generates the Octopus Java model based on the UML model. The code generator will add OCL expression by updating Java model based on the OCL model. At the end of code generation, the Octopus Java model will be translated into POJO.

The code generator consists of sub code generators which are controlled by *com.klasse.octopus.codegen.TransformationController*. Taking an object of *com.klasse.octopus.model.IPackages* (UML model) as an input, the code generator calls the *InvariantGenerator* (as a sub code generator) to update the Java model based on the invariant expressions.

Visitor pattern is used by code generator to update the Java model based on invariant expression. The visitor will visit all the nodes in the Java model tree and performs process when visiting the node. With Visitor pattern, the code generator can do a specific process without changing the structure of Java model. All visitors implements *IPackageVisitor*

interface which defines the visited Java model nodes such as Classifier (*IClassifier*), Package (*IPackage*), Interface (*IInterface*), etc. The visitor implementation defines the process while it visits the intended Java model node. In case *InvariantGenerator*, it defines an operation while visiting a Classifier (*IClassifier*) node.

The *InvariantGenerator* checks if there is an invariant expression inside a Classifier (*IClassifier*) node and if yes, the *InvariantGenerator* will take the OCL expression models and update Java model for each OCL expression model through *ExpressionCreator*.

3.2.5 OCL Invariants Expression

An OCL expression, for example an invariant *ofAge* , is formed by a few instances of few OCL expression model classes (to avoid ambiguity, these instances will be called just OCL model). The OCL models are linked together and can be looked as a tree which is called an Abstract Syntax Tree (AST).

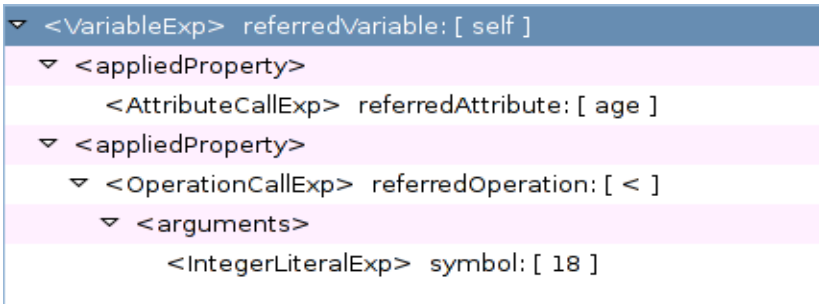

Figure 9: Abstract Syntax Tree of OfAge invariant

ExpressionCreator translates an OCL Invariant expression into a Java class method that has a *conditional* statement (*if then else* statement). The constraint of the invariant expression is placed inside the *condition statement*. The method name is "*invariant_*"+invariant name (e.g *invariant_ofAge*) or Octopus will generate the method name automatically if the invariant name is not defined. This method throws an *InvariantException* if the invariant is broken.

The *ExpressionCreator* will check the type of OCL expression and call the specific *ExpressionCreator* based on an order in Figure 10.

For example, there is an invariant named *ofAge* inside the *Customer*. The *InvariantGenerator* will call *ExpressionCreator* and put the *OCLExpression* object as an input. *ExpressionCreator* will call *getAppliedProperty()* in the *OCLExpression* object and based on *OCLExpression* type, it will call the related expression generator and starts to update the Java model.

Figure 10: OCL Metamodel In Octopus

In *ofAge* invariant, started by *VariableExp*, *ExpressionGenerator* calls *makeVariableExp* and subsequently calls *getAppliedProperty()* in *OCLExpression* object to get sets of applied properties, which are *AttributeExp*, *OperactionCallExp*, *IntegerLiteralExp.*

All the sub creators return the Java codes that construct the conditional part of *If statement*. *VariableExp : self* is translated into *this* in the Java code, *AttributeCallExp : age* is translated into *getAge()*. *OperactionCallExp: referredOperation :[<]* istranslated into < and *IntegerLiteralExp : symbol:[18]* into *18* .

Listing 12 shows the Java code generated by the Octopus code generator.

```
/** Implements self.age < 18
 */
public void invariant_ofAge() throws InvariantException {
      boolean result = false;
      try {
            result = (this.getAge() < 18);
      } catch (Exception e) {
            e.printStackTrace();
      }
      if ( ! result ) {
            throw new InvariantException(this, message);
            ... message error
      }
}
```

Listing 12: Java code for ofAge invariant generated by Octopus

Let's take another invariant example : "All card of customer must be valid" [Listing 13].

```
context Customer
      inv allCards : cards->forAll(valid=true)
```

Listing 13: Customer::allCard invariant

For a more complex OCL Expression such as *forAll()* , *ExpressionCreator* calls *LoopExpCreator*.

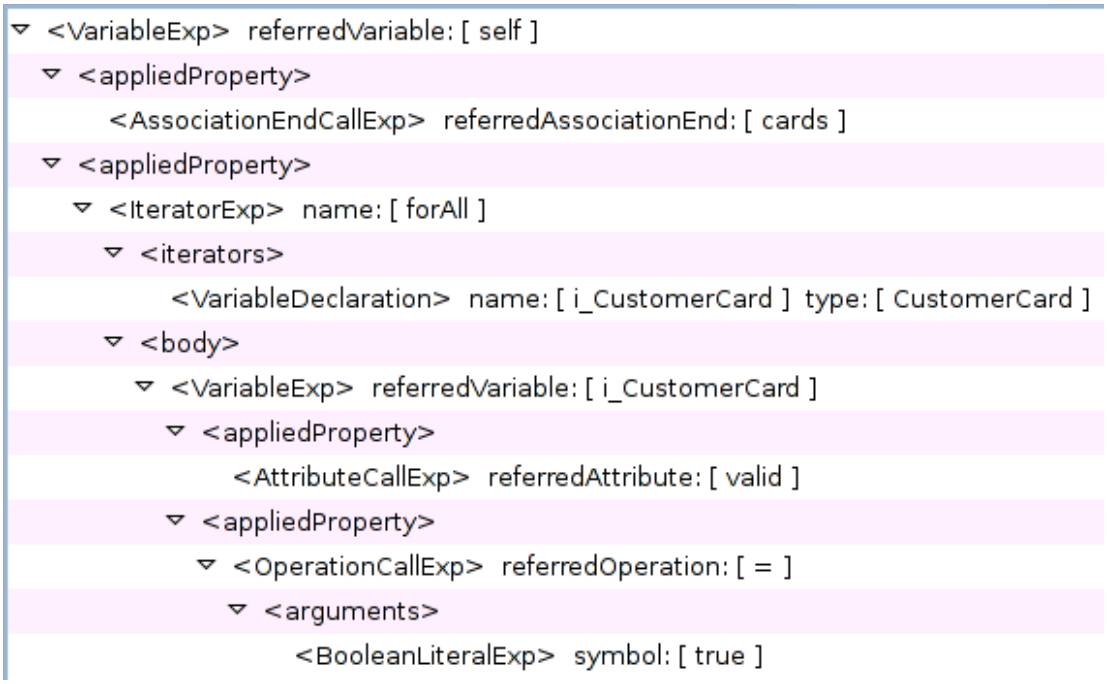

Figure 11: Abstract Syntax Tree of allCards invariant

LoopExpCreator creates the *forAll1()* method based on information stored inside *IteratorExp* object [Figure 11] . It creates *iteration* with *i_CustomerCard* as a variable whose type is *CustomerCard* and a *loop operation* which each *loop*, *i_CustomerCard* will compare its *valid* attribute equals with *true*. This generated methods returns boolean because *forAll* in essence is a *boolean* expression (either all or not all).

```
private boolean forAll1() {
      Iterator it = this.getCards().iterator();
      while ( it.hasNext() ) {
            CustomerCard i_CustomerCard = (CustomerCard) it.next();
            if ( !(i_CustomerCard.getValid() == true) ) {
                  return false;
            }
      }
      return true;
}
```

*Listing 14: Java code for **forAll(valid=true)** expression generated by Octopus*

As a result, the *forAll11* method is generated as an intermediate method to process the *iterator* expression (in this case it is *forAll1()*) [Figure 13].
An the end, *f invariant_allCards()* calls *forAll1()* method to perform invariant checking.

```
/** Implements self.cards->forAll(
i_CustomerCard : CustomerCard | i_CustomerCard.valid = true )
*/
public void invariant_allCards() throws InvariantException {
      boolean result = false;
      try {
            result = forAll1();
      } catch (Exception e) {
            e.printStackTrace();
      }
      if ( ! result ) {
            String message = "invariant allCards ";
            message = message + "is broken in object '";
            message = message + this.getIdString();
            message = message + "' of type '" +
                        this.getClass().getName() + "'";
            throw new InvariantException(this, message);
      }
}
```

Listing 15: Java code of allCards invariant generated by Octopus

3.3 Summary

Octopus is one of MDA tools that use UML and OCL languages (PIM) to define business models and business rules. Octopus has its own UML and OCL meta-model and Java PSM meta-model. During the UML and OCL syntax parsing process, Octopus instantiates the UML and OCL models based on the UML and OCL expressions. It uses *visitor pattern* to visit and do process inside the specific visitable node (Java PSM). This pattern allows the visitor classes to visit and do some process inside visited node without changing visited nodes structures.

Some works have been done in Octopus to transform OCL invariant into domain specific languages by defining new domain specific language PSMs.

4 Run Time Invariant Checking

The automatic business rule checking during runtime is highly desirable. Consider an enterprise application which deals with many business objects holding a business rule, an action can change the business object and make the business object inconsistent.

We need the way not only to detect the broken business rules, but also how to deal with the broken business rules. In other words, a runtime invariant checking is necessary to preserve the model's consistency.

Let's consider another example. In MDA we are dealing with PSM such as SQL. SQL-PSM developers define SQL model so that an invariant can be mapped into the database layer. SQL-PSM will be transformed into SQL statements. A user must run the generated SQL statement into database and see if it works. PSM-SQL developers can define the constraints inside their model in order to get an the error free or well formed SQL syntax. In this case, runtime checking is needed to ensure that see the PIM-into-PSM transformation fits SQL-PSM constraints. With run-time invariant checking, any model creation can be checked if it fulfills its own constraints without going into the model implementation.

As one of MDA tools, besides generating Java code for the UML model, Octopus also generates the Java code that reflects OCL invariant expressions. A user can call the *invariant_ofAge()* method to check if this object breaks the *ofAge* invariant.

When a class is instantiated and it becomes an object in a runtime , users can change the object attribute values and these changes can break one of the object constraints. Meanwhile the constraints must be true or satisfied during its life time or must be consistent. It is why consistency checking is needed.

The easier way is to call the *checkAllInvariants()* method after user changed any value. But it will run all the invariant checking of an object regardless of the possibility that some invariants might not be affected by the value change.

4.1 Invariant Checking in ILOG

A rule based system can be used to preserve the invariant during the object life time. An invariant can be translated into a *rule*. JRule rule engines will try to find which *rule* can be satisfied by the present *fact* in the working memory. *ofAge* invariant defines that all Customer must have *age < 18* . Inside condition part of *testAge* iLog rule, *age*>=18 is set as the reverse

condition of *age <18,* because we want to detect the *Customer* object that breakes this invariant. Listing 15 is the example code of how to preserve the *allCards* invariant in JRule rule.

The advantages of using property rule-based system are it is already a integrated system (from rule authoring until rule testing), and it has implemented the highly customized Rete algorithm, which can speed up the proses of rule matching. JRule also has the *rule conflict resolution* in case one *rule* cancels another *action* activation from the *agenda*.

One of its disadvantages is that it has its own rule language (rule structure and special key word). It is also bound into Java programming language. All *facts* must be presented in the Java classes and a rule developer must have at least basic knowledge in Java.

```
rule allCards {
      when {
            ?cus : Customer();
            exists ?CustomerCard (valid==false;owner=?cus);
      }
      then {
            System.out.println("allCards invariant is broken");
      }
};
```

Listing 16: allCard Invariant in JRule rule

There is an option to use JRule rule engine to preserve the invariant. Using Octopus, all business objects (UML model) which are represented in PIM can be translated into Java PSM and later they are inserted into the JRule rule engines. The OCL invariant (PIM) can be translated into Domain Specific Language (DSL) which is JRule rule using the transformation tool.

4.2 Invariant Checking in POJO

Another approach is to deal with the transformation tools that transform PIM into PSM in Octopus. Additional process or transformations are added inside *ExpressionCreator* especially the *InvariantGenerator*.

The idea is to make object call the invariant checking method after any value change which can break the invariant. In Java, a user can change the object attribute value through object

setter that calls the invariant checking method. For example *setAge()* method calls *invariant_ofAge()* method automatically after the *setAge()* changed *age* value.

The DSL developers also can use Octopus to generated Java code for their DSL meta-models . And these meta-model codes can be used by Octopus to transform PIM into PSM. Transformation itself is instantiating DSL meta-model classes. After instantiated, these objects automatically can call the invariant checking method to check their constraints. This automatically invariant checking can help the DSL developer to get the "well-formed" syntax based on the constraints inside the DSL meta-model.

In the same time in this project, the new rule meta-model based on the proprietary rule languages will be defined to model the OCL invariant instead of using the defined OCL meta-model. There is a pre-assumption that OCL expression has similarities with rule expression. This step wants to see the possibility whether the OCL meta-model can be reused to define rule in rule-based system (or let's say it is enough to use OCL language). A new transformation tool will be defined to transform the Rule meta-model PIM into Java PSM. The process will be started based on invariant expression from the simplest into the more complex one.

4.2.1 Customer::ofAge invariant

ofAge invariant involves the *age* attribute and the Customer classifier. Inside *rule*, a classifier can be used as a condition. *Customer()* means if there is an object of *Customer* in *working memory*, the *ClassifierExp* is made to present that condition. AttributeExp::[age], OperationExp::[>=] and IntegerExp::[18] express the rest of the expression.

Below is the Java code created by *RuleGenerator*.

```
public class Customer{

	public void setAge(int element) {
		if ( f_age != element ) {
			f_age = element;
		}
		this.ofAge();
	}

	public void ofAge() {
		if(this.getAge()<18){
			System.out.println("ofAge is Broken");
		}
	}
}
```

The prototype of the rule meta-model is defined based on *ofAge* invariant. The *Rule meta-model* object will be instantiated manually. The *RuleGenerator,* which implements *IPackageVisitor* interface, takes the UML model and Java model as inputs. When visiting *IClassifier* node inside *IPackage*, The *RuleGenerator* will call *ClassifierCreator*, *AttributeCreator*, *OperationalCreator*, and *IntegerCreator* to update the Java model

Figure 12 shows the UML diagram of the rule meta-model

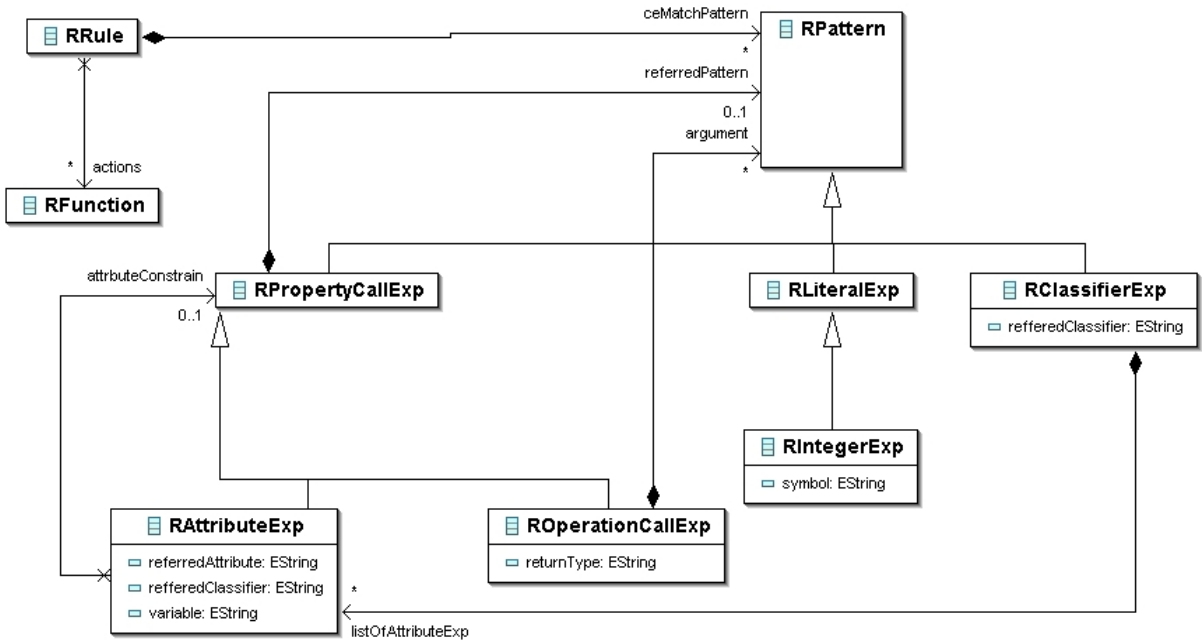

Figure 12: First of Rule Meta-model prototype

.

4.2.2 Customer::cardSize Invariant

The next invariant example is *cardSize* which involves an *association* between *Customer* and *CustomerCard* classes.

```
context Customer
     inv cardSize : cards->size()>=5
```

Listing 17: allCards Invariant

cardSize imposes that an object of *Customer* must have *CustomerCard* objects more than or equal to 5. *cards* is the navigation from the *Customer* to the *CustomerCard*. Before adding a new rule expression meta-model, this is the rule *cardSize* expressed in JRule rule.

```
rule cardSize {
      when {
            ?cus : Customer();
            ?cards : Collect( CustomerCard (owner=?cus) );
            evaluate(?cards.size()<5);
      }
      then {
            System.out.println("allCards invariant is broken");
      }
};
```

Listing 18: cardSize invariant in JRule rule

A new *VariableExp* which refers to c*lassfifier* is defined. *CollectExp* defines collecting an object based on *ClassifierExp* and *AttributeExp*. *CollectExp* always returns a collection. *EvaluateExp* accomodates the expression that returns *boolean*. *size()* is expressed by *OperationExp*.

Octopus generates the additional method to express the *association*. The degree of *association* (*toMany* or *toOne*) influences the *setter*. *toMany association* creates two additional *setters* which are *addToXX()* and *removeFromXX()*methods, while *toOne association* creates *setXX()* method. The additional code which calls the invariant will be added inside these setters as well as additional Octopus generated methods for *association* (*z_internalRemoveFromCards(), z_internalAddToCards()*).

There is a case when an object of *CustomerCard* changes its *owner*, it potentially brakes the *allCard* invariant. When changing its *owner*, the object of *CustomerCard* will call *z_internalRemoveFromCards()* which will make the previous *owner* cut its reference with current *CustomerCard* object and *z_internalAddToCards()* which will make the new *owner* set a reference to the current *CustomerCard* object.

The following Java code is a set of methods with additional codes generated by *RuleGenerator* :

```
public class Customer{

      public Set collect_CustomerCard() {
            Set tempSet = new Set();
            for (Iterator it =
                  CustomerCard.allInstances().iterator();
                  it.hasNext();) {
                        CustomerCard el = (CustomerCard) it.next();
                        if (el.getOwner().equals(this)){
                          tempSet.add(el)
                        }
            }
            return tempSet;
      }


      public void cardSize() {
            if(this.collect_Customer().size()>5){
                  System.out.println("ofCards is Broken");
            }
      }
      public void addToCards(CustomerCard element) {
      ...
            this.f_cards.add(element);
            this.cardSize();
      ...
      }
      public void z_internalAddToCards(CustomerCard element) {
            this.f_cards.add(element);
            this.cardSize();
      }
      public void z_internalRemoveFromCards(CustomerCard element) {
            this.f_cards.remove(element);
            this.cardSize();
      }
      public void removeFromCards(CustomerCard element) {
      ..
            this.f_cards.remove(element);
            this.cardSize();
      ..
      }
}
```

The rule meta-model will be updated with additional *VariableExp*, *CollectExp*,*OperationExp* meta-models.

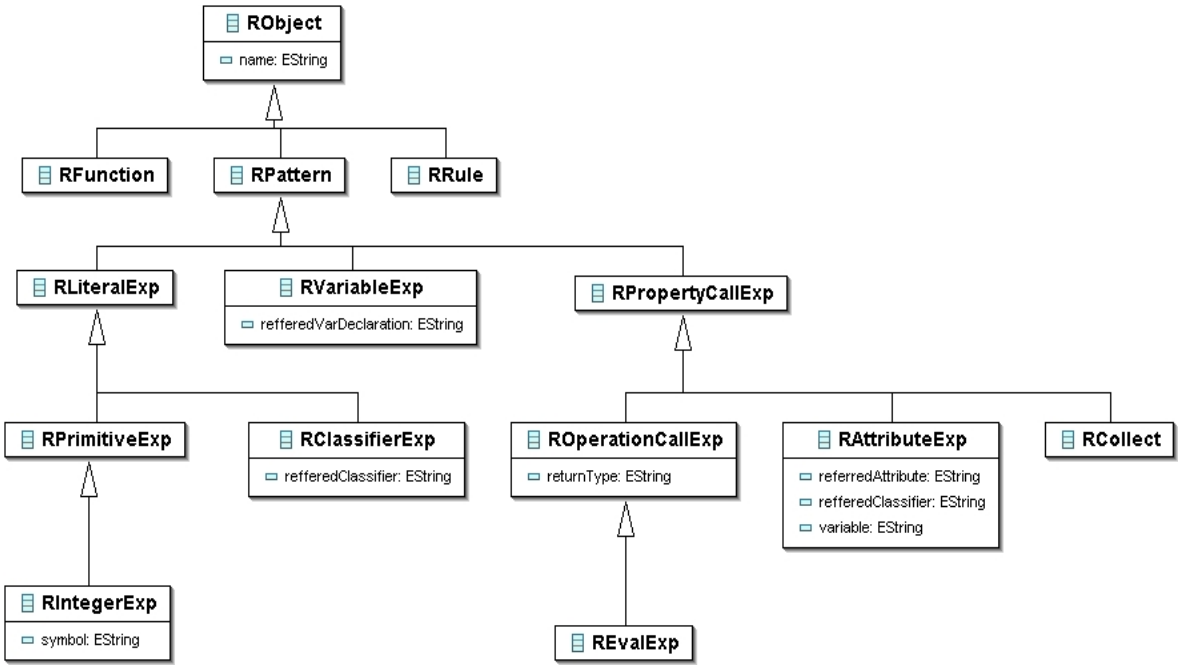

Figure 13: Rule meta-model prototype 2

4.2.3 Transaction::ofAge invariant

In the last 2 attempts, a new Rule meta-model is defined and all related Rule expression generators are made to translate Rule PSM into Java meta-model. After some efforts to model the rule language, it comes to a (pre) conclusion that new defined rule meta-model goes like the OCL meta-model or OCL meta-model can be reused in rule meta-model because still there are some differences between a rule and an OCL expression.

In this approach, instead of defining a new meta-model, OCL meta-model will be used and the same Java code will be produced and just the additional process in code generator will be added.

For a run time checking for an invariants that involves *association* in two or more classes, a non context object must navigate into the context class object and call the invariant checking method from the context object.

```
context Transaction
     inv ofAge : card.owner.age >18
```

Listing 19: Transaction::ofAge invariant

If an object of a *Customer* changes its age *attribute*, it has to navigate to its *context* object, which is an object of *Transaction*, and call the invariant checking method from *Transaction* object. When navigating to the context class, the degree of association has to be considered. In this case, an object of *Customer* must iterate over a collection of *Transaction* objects and call the *Transaction* invariant checking method. Taking the OCL expression [Figure 14] as an input, the updated *InvariantGenerator* will generate the additional code to do the navigation.

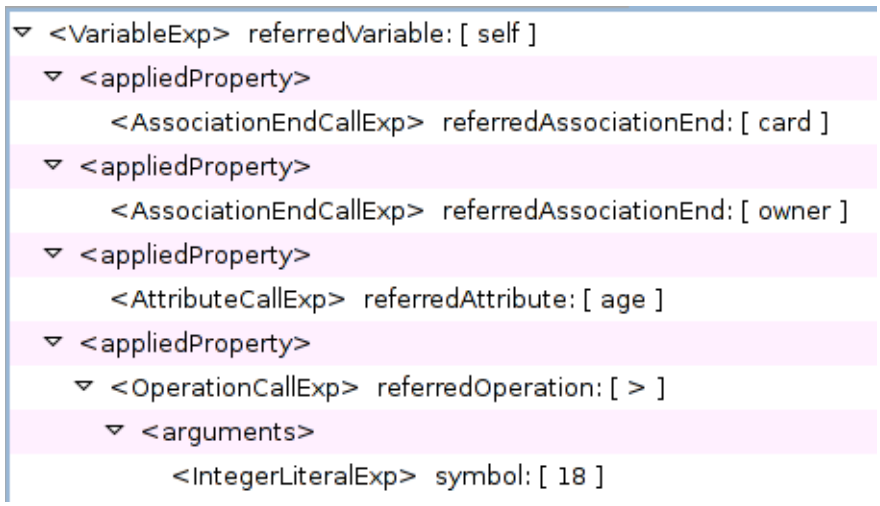

Figure 14: The Transaction::ofAge Abstract Syntax Tree

OCL expression has *def* expression to define an additional operation inside the UML model. *Meng Xue* in his master thesis proposes "Back Navigation Algorithm" to generate a *def* OCL expression from existing OCL invariant expression that involves the *association*.

This *def* OCL expression does the navigation to the context object and retrieves the context objects [MXE01]. In short the *back navigation algorithm* implementation is a visitor that visits all OCL invariant nodes, saves *AssociationEndCallExp* and *AssociationClassExp*, and builds the *back path navigation.*

From the *Transaction::ofAg*e OCL expression the new operations are generated inside *def.ocl* file [Listing 20].

```
context randl::Customer
def : getContext_OfAge()
     : Bag(Transaction)
     = cards.transactions

context randl::CustomerCard
def : getContext_OfAge()
     : Set(Transaction)
     = transactions
```

Listing 20: def OCL expressions generated by back path navigation

Later by the Octopus *ExpressionGenerator*, these *def* operation is translated into the following code in Java *Customer* class :

```
public class Customer{

     public List<Transaction> getContext_OfAge() {
          return collect1();
     }

     private List<Transaction> collect1() {
          List /*(Transaction)*/ result =
               new ArrayList( /*Transaction*/);
          Iterator it = this.getCards().iterator();
          while ( it.hasNext() ) {
               CustomerCard i_CustomerCard
                         = (CustomerCard) it.next();
               Object bodyExpResult =
                    i_CustomerCard.getTransactions();
               result.addAll( (Collection)bodyExpResult );
          }
          return result;
     }
}
```

If an object of *Customer* changes its *age* attribute, it must call the *getContext_OfAge()* method and iterate over the list of *Transaction*,and then call the *Transaction* invariant checking method which is generated automatically by *InvariantGenerator*.

```
public void setAge(int element) {
      if ( f_age != element ) {
            f_age = element;
      }
      Iterator it = this.getContext_OfAge().iterator();
      while ( it.hasNext() ) {
            Transaction t = (Transaction) it.next();
            t.invariant_ofAge();
      }
}
```

Listing 21: setAge() with additional Transaction::ofAge invariant checking

The *ExpressionGenerator* will be extended to deal with the degree of *association*. This generator will add additional code to call the invariant checking method over a collection or a single object. The *InvariantGenerator* will call Java code creators based on OCL expression to add Java code into Java model. Taking the same OCL expression, a new expression creator (*testCreator*) uses the following algorithm to add the additional code inside the Java model.

```
many = false
makeExpression(OCLExp,invariantName){

If AssociationEndExp or AssociationClassEndExp
      get the associated end variable name
      get the other end association type (classifier or collection)

      if classifier
            construct Java code :
                  getContext_XXX().invariant_xxxx().
            add code inside setter of variable name (plus additional
            association methods)

      if collection
            construct Java code :
                  iterate over item from getContext_XXX()
                  each item do invariant_XXX()
            add code inside setter variable name (plus additional
            association methods)
            many=true

If AttributeExp
      if many
            construct Java code :
                  iterate over item from getContext_XXX()
                  each item do invariant_XXX()
            add code inside attribute setter
      else
            construct Java code :
            getContext_XXX().invariant_XXX()
            add code inside attribute setter
}
```

The invariant checking method will be called when there is a change in attributes and in the *attribute* that maps *association*. These following methods will trigger *Transaction.invariant_OfAge()* method :

- *Transaction.invariant_ofAge()*
- *Transaction.setCard()*
- *CustomerCard.addToTransactions(),*
- *CustomerCard.removeFromTransactions()*
- *CustomerCard.setOwner()*
- *Customer.addToCards()*
- *Customer.removeFromCards()*
- *Customer.setAge()*

Listing 22 shows generated Java code inside *CustomerCard* class.

```
public Set<Transaction> getContext_OfAge() {
      return this.getTransactions();
}
```

Listing 22: additional method to retrieve the Transaction object as a context object

Listing 23 shows generated Java inside *setOwner()* method.

```
public class CustomerCard{
      public void setOwner(Customer element) {
            if ( this.f_owner != element ) {
                  if ( this.f_owner != null ) {
                        this.f_owner.z_internalRemoveFromCards(
                              (CustomerCard )this );
                  }
                  this.f_owner = element;
                  if ( element != null ) {
                        element.z_internalAddToCards(
                              (CustomerCard)this );
                  }
            }
      /** the additional code **/
            Iterator it = this.getContext_OfAge().iterator();
            while ( it.hasNext() ) {
                  Transaction t = (Transaction) it.next();
                  t.invariant_ofAge();
            }
      /** end additional code**/
      }
}
```

Listing 23: additional Java code inside setOwner()

```
public class CustomerCard{

      public void z_internalRemoveFromOwner(Customer element) {
            this.f_owner = null;

      /** the additional code **/
            Iterator it = this.getContext_OfAge().iterator();
            while ( it.hasNext() ) {
                  Transaction t = (Transaction) it.next();
                  t.invariant_ofAge();
            }
      /** end additional code**/
      }

      public void z_internalAddToOwner(Customer element) {
            this.f_owner = element;

      /** the additional code **/
            Iterator it = this.getContext_OfAge().iterator();
            while ( it.hasNext() ) {
                  Transaction t = (Transaction) it.next();
                  t.invariant_ofAge();
            }
      /** end additional code**/
}
```

Listing 24: Additional Java code inside Octopus additional methods

Customer and *CustomerCards* objects must navigate to the context object whose type is *Transaction,* and from this *Transaction* object, it will navigate to *Customer* object through *CustomerCard* to check if the attribute *age* is greater than *18*. Even though it has not been proven yet (statistically or mathematically), this approach is inefficient. It must do navigation back and forward through the object to check if the invariant is broken.

In rule based system that implements the Rete algorithm, only the *LiteralNode* that stores the condition *age<=18* will update its *AplhaMemory*. For example an object of *Customer* changes *age* attribute from *17* (which breaks the *transaction::ofAge* invariant) to *20*. The *LiteralNode* will discard this from its *AlphaMemory*. Later *LiteralNode* [*age<=18*] will pass all objects in its *AlphaMemory* to all nodes below which triggers rule reevaluation.

Another disadvantage of implementing OCL invariant directly into code is that it can't handle additional action if there is any inconsistency. In Octopus generated code, a invariant method simply returns the *InvariantException*. It means a system that uses this class must do exception handling (with *try catch* statement) to deal with *inconsistency*. By Applying rule-based system, *inconsistency* can be handled easily inside *action* part. The application and the

rule itself are *loosely* coupled. In practice, a software developer can add new rules during *runtime* or remove the old *rules* during the *runtime*. Compared to with Octopus invariant checking code, a new code must be generated based on the newly defined *invariant* and the system must use the new Java classes and remove the old ones which do not apply the new invariant.

4.2.4 Production Rule Representation

Object Management Group has proposed the draft of the Production Rule Representation (PRR), which addresses the need for modeling the production rule. This draft specifies that the target platform as a rule based system that uses forward chaining method specially implementing Rete algorithm.

One of the purposes of this draft is to speed up the adoption of production rule component in every software system and allow interoperability between rule based system implementations.

Stating that UML isn't enough to define the business rule, it argues the need of constraint expression (OCL) to represent business rule. Below is the PRR meta model suggested by OMG :

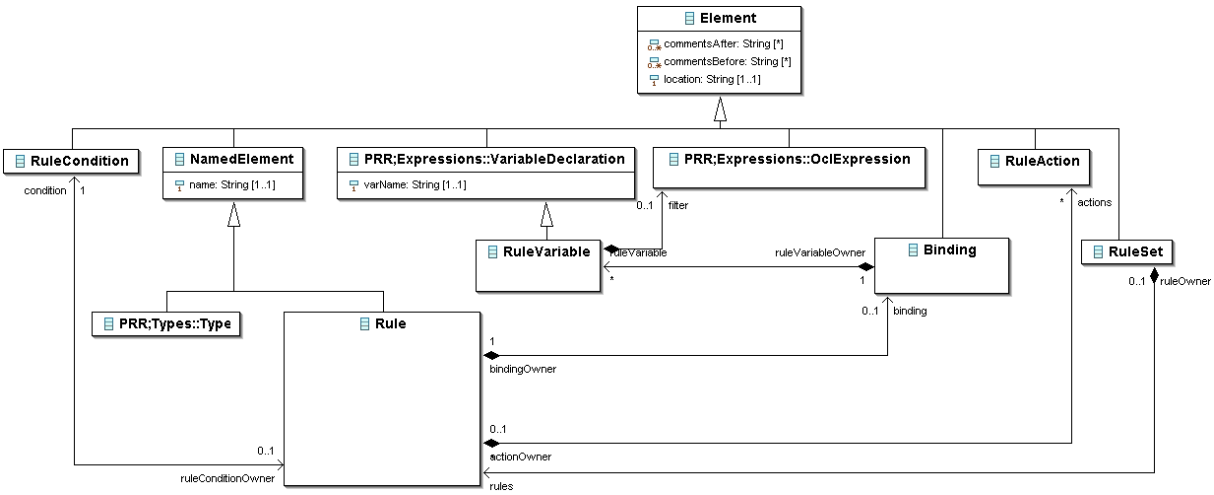

Figure 15: Production Rule Representation

From Figure 15, OCL expression is used as *filters* to represent the constraints. This PRR supports the pre-assumption that OCL expression can be reused inside *rule* meta-model instead of defining a completely new *rule* meta-model.

4.2.5 Eclipse Modeling Framework

Eclipse Modeling Framework is Java framework and code generation facility for building tools and other applications based on structured model [EMF01]. As an MDA tool, it helps software developers to define their software structure based on model and generates the Java code in an efficient, correct, and easily customized manner. A model can be defined using the annotated Java, XML, and XML Meta-data Interchange (XMI).

As a framework, EMF offers several capabilities to help in modeling software, which are serializing from and into XML file, graphically editing and manipulating models. It offers run time capabilities such as change notification, persistence support, and reflective API.

4.2.5.1 Generated Java Code

EMF divides the generated Java code based on models into three packages : *xxx* (stands for the name of the model package like *randl*), *xxx.impl*, *xxx.util*. The *util* package consists of the *Adapter Factory* and *Adapter classes*. And the *xxx* package contains all Java *interfaces* that contains the *setter* and *getter* for each *attribute* and *reference* of the corresponding model class [EMF01]. The *xxx.impl* consists of all implementation classes that extend *EObject* which has capabilities to participate in EMF *notification* framework.

4.2.5.2 EMF Adapter

EMF *adapters* are what other programing language frameworks called *observer*. It is called *Adapter* because instead of observing changes, it can extend the model behavior without changing the model structure [EMF01]. A generated Java class can be attached with an *adapter*. An additional behavior can be added inside the *adapter,*, that will responds if there are changes inside attached class. While generating the Java code, EMF also generates the *AdapterImpl* which has *public void notificationChanged(Notification notification)* method. The additional behaviors are implemented inside the *adapter implementation* class which extends the *AdapterImpl* class. Beside observing changes, an *adapter* can also find out what has been changed such as *age* attribute in *Customer* class from *Notification* class which is passed as an argument in *notificationChanged* method [Listing 25].

```
public class CustomerAdapter extends RandlAdapterFactory {
      public void notifyChanged(Notification notification){
            switch (notification.getFeatureID(Customer.class)){
            case RandlPackage.CUSTOMER__AGE:
                  if((Integer)notification.getOldValue()<=18)
                        System.out.println(
                              "Customer::ofAge is broken");
            }
      }
}
```

Listing 25: CustomerAdapter example in EMF

4.2.5.3 Octopus Into Emfatic Plug-in

Emfatic format is human-readable textual notation for EMF model. In essence, it represents *ecore* model in textual format. Works to transform invariant from Octopus into Emfatic have been developed [JIB01]. This project extends Octopus into Emfatic plug in to include OCL expression, such as invariant into Emfatic annotation. Using JET, the Java code that reflects the OCL invariant expression, derived attributes and references, and operations can be generated without requiring any post generation custom code [ECL02].

4.3 Summary

The run-time checking can be done in POJO. An extended transformation tools (in Octopus) will create additional methods for *associations* and doing navigations to the *context* class. This approach is inefficient because non context object must navigate to the context object and call the invariants checking from context object.

In contrast, EMF offers some features through its framework. There is a possibility to implement Rete algorithm to speed up the runtime model checking based on the model constraints.

OCL meta-model can be reused inside the *rule* meta-model to express rule constraints or conditions. It saves developer efforts to define a completely new rule meta-model.

5 Summary And Outlook

MDA emphasis high level abstraction or modeling throughout the software development process. Not only well formed languages like UML and OCL are needed to create a model but also tools that support the modeling process. From model authoring to transformation between PIM and PSM are done automatically by modeling tools. A precise model is needed to accommodate the modeled system and precise model makes the transformation possible.

PSMs have their own constraints and transformation from PIM into PSM must confirm PSM constraints. In many case, MDA tools don't offer run-time constrains checking while doing transformation from PIM to PSM. The constrains checking usually must be done by running the generated platform codes.

Run time constraints checking is highly desirable in order to know that the transformation from PIM to PSMs meets PSM constraints immediately. It helps PIM into DSL transformation to achieve "well formed syntax" in the transformation level without going into specific DSL implementation platform. In business application, invariant runtime checking guarantees that each business object meets its business rule and the system offers future actions if there are business rule violations.

In this project, it is shown that runtime checking in platform specific code is inefficient meanwhile a rule-based system which implements Rete algorithm offers capabilities to perform runtime checking in better way. Using the proprietary rule-based system, OCL invariant expression can be transformed into Rule DSL and UML model can be generated into specific rule engine implementation platform. Another approach is to implement the Rete algorithm inside the modeling tools such as EMF. Further works can be done by implementing Rete algorithm inside MDA tools such as EMF. Octopus to Emfatic invariant transformation and EMF notification framework offer support to the Rete implementation. So that run-time checking can be done in the modeling level.

During the project, it is shown that the OCL expression has almost the same characteristic compared with the rule characteristic and OCL expression has been used by Production Rule Representation to express constraints. PRR tries to make meta-model for business rules and uses OCL to define the constraints. In this case OCL meta-model can be reused into another PSM like *rule* in rule-based system.

Bibliography

[CGW01] Modeling Rule-Based Systems with EMF.November. 2004. October. 2006. [http://www.eclipse.org/articles/Article-Rule%20Modeling%20With%20EMF/article.html].

[DRL01] DROOLS Documentation.September. 2006. October. 2006. [http://labs.jboss.com/file-access/default/members/jbossrules/freezone/docs/3.0.3/html_single/index.html].

[ECL01] The Eclipse Modeling Framework (EMF) Overview.June. 2005. October. 2006. [http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html].

[ECL02] Implementing Model Integrity in EMF with EMFT OCL.August. 2006. October. 2006. [http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html].

[EMF01] Budinsky. Frank, et all. Eclipse Modeling Framework: A Developer's Guide. Addison Wesley, 2003.

[EMF02] Daly, Chris. Emfatic Language Reference (draft). 2004.

[EMF03] Implementing Model Integrity in EMF with EMFT OCL.August. 2006. October. 2006. [http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html].

[JAV01] Weiss, Mark Allen. Data Structure And Algorithm Analysis In Java. Addison Wesley, 1999.

[JES01] Hill, E Friedman . Jess In Action. Manning, 2003.

[JIB01] Shidqie, A. Jibran. Conversion of Octopus UML Models Into Eclipse UML2 Models. Technische Universität Hamburg Harburg. 2006.

[MDA01] Kleppe Anneke, Jos Warmer, Wim Bast. MDA Explained: The Model Driven Architecture™: Practice and Promise. Addison Wesley, 2003.

[MXE01] Xue, Meng. Prototype for Solving Consistency Problems in Model Engineering. Technische Universität Hamburg Harburg. 2006.

[OCL01] Kleppe Anneke, Jos Warmer. Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition. Addison Wesley, 2003.

[OCL02] Giese, Martin, Reiner Hähnle, Daniel Larsson. Rule-based Simplification of OCL Constraints. 2004.

[OCL03] Octopus. October. 2005. October. 2006. [http://www.klasse.nl/octopus/index.html].

[RET01] The RETE Algorithm.December. 2005. October. 2006. [http://www.cis.temple.edu/~ingargio/cis587/readings/rete.html].

[RET02] Rete: Language And Mind.October. 2003. October. 2006. [http://www.ai.mit.edu/courses/6.034b/recitation6.pdf#search=%22rete%20language%20and%20mind%2BMIT%22].

[VNT01] Tedjasukmana, Veronica N. Translation of OCL Invariants into SQL:99 Integrity Constraints. Technische Universität Hamburg Harburg. 2006.

[WFL01] Explaining RETE.April. 2006. October. 2006. [http://woolfel.blogspot.com/2006/04/explaining-rete.html].

[WIKI01] The Visitor Pattern.September. 2006. October. 2006. [http://en.wikipedia.org/wiki/Visitor_pattern].

Appendix A : The Royal And Loyal Class Diagram