Prototyping an Infrastructure for MDA

Kai Yuan

Submitted in partial fulfillment of the requirements for the degree Master of Science in Information and Media Technologies.

supervised By:
Prof. Dr. Ralf Moeller (STS)
Prof. Dr. Friedrich H. Vogt (Telematik)

Hamburg University of Science and Technology
Software Systems Institute (STS)

Abstract

Model Driven Architecture (MDA) is getting popular with its particular development process in software development. Octopus, an OCL expression checking tool, can be applied as MDA transformation engine to build an Octopus based MDA infrastructure. This thesis explains how Octopus works as transformation engine in MDA infrastructure and also presents two prototypes of Octopus based MDA infrastructure. One is MDA infrastructure with UML statechart as input, the other one is for generating JBoss based Web Service security deployment descriptor which is the case that MDA used in an enterprise architecture.

Declaration

Hereby I declare that this project 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, 5th Sep. 2005
Kai Yuan

Acknowledgement

I would like to thank Professor Ralf Moeller of STS for supervising this thesis and being very helpful with finding a topic. Thanks also go to Professor Friedrich H. Vogt of Telematik department for being the co-corrector.

Mr. Miguel Garcia of STS was very patient in providing advice and direction on the thesis, thanks also go to him.

Table of Contents

List of Figures

1 Introduction

1.1 Motivation

Guidelines for UML profile definition have been established by the OMG, and a growing number of sophisticate profiles are being proposed (for QoS, EAI, Product Lines, Agent Based Systems). Such profiles are valuable because they embody best-practices about the software architecture they describe. However, the OMG guidelines and the profile descriptions are not expressed in terms of a particular modeling infrastructure, given that none has gained reference status. Given that Octopus has not only metamodeling support (as Eclipse's EMF has) but also OCL support, the OMG guidelines for implementing profiles and the changes needed to the UML2 metamodel for a particular profile will be materialized in terms of Octopus models. Therefore it is significant to build modeling infrastructure with Octopus.

In this thesis two MDA infrastructure prototypes will be designed and implemented.

- Statechart based MDA infrastructure prototype

Most business logic work flows or business protocols are playing very important role in an enterprise application. Usually they can be modeled as UML statechart. For example, the business protocol of BPEL4WS can be modeled in statechart. A MDA infrastructure with statechart as input model can output various target models, for example, Java model, which can generate java source codes at the end. Therefore this MDA infrastructure prototype can solve a set of problems with startchart as input model so that simplified the development of business logic workflow.

- Apply Octopus based MDA infrastructure on an enterprise architecture.

In web-based enterprise applications, any business components are deployed on a application server. And application servers today are getting powerful; they can control the data persistent rule, manage transaction or manage security mechanisms (for example authentication and authorization) via configurations or deployment descriptors. Developers benefits a lot from those application server provided features, however, the configurations and deployment descriptors are different software artifacts from those components which

developers concentrate on. So it would be nice if there is a MDA infrastructure that can from a certain model, for example secureUML model for access control security, generate the configuration codes automatically. This MDA prototype will get an input of secureUML model and output the web service authorization concerned deployment descriptors snippet automatically.

1.2 Structure of this thesis

In this chapter why we need an Octopus based MDA infrastructure is discussed. And the two prototypes this thesis will bring are introduced briefly as well. Next chapter will introduce some background technologies and concepts used in this thesis, such as MDA and Octopus. In chapter three, the design and implementation of statechart based MDA prototype is going to be explained. Chapter four will introduce how Octopus based MDA infrastructure is used in an enterprise architecture. A prototype that getting a secureUML model and EJB model as input generates a web service deployment descriptor on security issue is analyzed as well. Finally, conclusion is given in the chapter five.

2 Background

Some technologies and concepts that are used throughout this thesis are introduced and explained in this chapter. They are MDA (Model Driven Architecture), OCL (Object constrain Language) and Octopus (OCL Tool for Precise UML Specifications).

2.1 MDA (Model Driven Architecture)

The Model Driven Architecture (MDA) is a framework for software development defined by the Object Management Group (OMG). Model is one of the most important keys for MDA. In an MDA setting, modeling software system drives the software development process.

2.1.1 MDA software development process

There is no big difference from traditional software development process at the beginning of MDA process: getting requirements and analyzing them.

Then, a model with a high level of abstraction that is independent of any implementation technology will be built [KWB03]. This model is called Platform Independent Model (PIM), it describes a software system that supports some business. In a PIM, no technical details are defined, for example, if a relational database will act as the data store, or whether EJB and Web services will be used at all in the system.

Next, the PIM is transformed into one or more Platform Specific Models (PSMs). A PSM tailored to specify the system in terms of the implementation constructs that are available in one specific implementation technology. For example, a relational database PSM can contain “table”, “column” or “primary key”, a Java PSM would have specific terms like “Class”, “interface” or “method/operation”. One PIM can be transformed into a number of PSMs, it depends on how many implementation technologies are needed.

The last step is to transform a PSM to code.

The software development lifecycle following MDA guidelines is represented below in Fig 2.1:

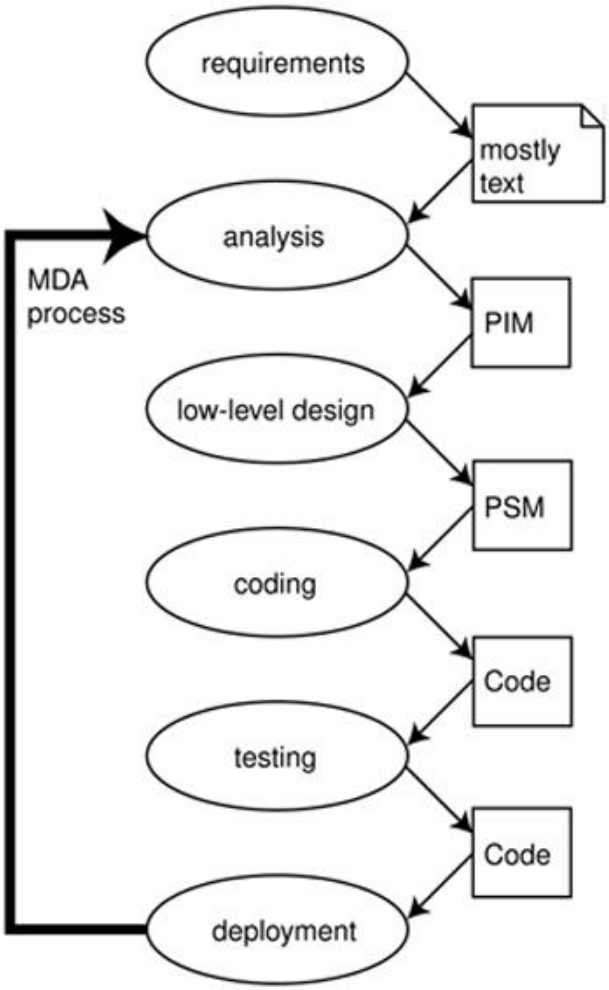


Fig 2.1 MDA software development process [KWB03]

2.1.2 Transformation

As we can see in the full MDA process lifecycle, transformation is the most important part, it directly influences the correctness of final codes. In this section, MDA transformation will be explained briefly.

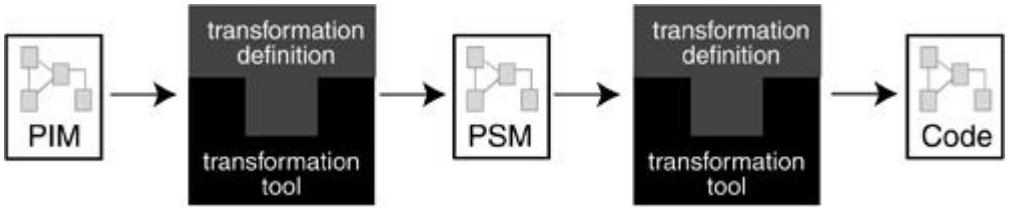


Fig 2.2 Transformations in MDA process [KWB03]

There are two transformations in MDA process, see Fig2.2. One is from PIM to PSMs, and another one is transforming PSM into final codes. Because a PSM has already closely fit its implementation technology, the second transformation is rather easy. However the first one is more complex.

Transformation Definition

In general, we can say that a transformation definition consists of a collection of transformation rules, which are unambiguous specifications of the way that (a part of) one model can be used to create (a part of) another model. Based on these observations, we can now define transformation, transformation rule, and transformation definition. [KWB03]

A transformation is the automatic generation of a target model from a source model, according to a transformation definition.

A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.

A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

The picture (Fig 2.3) shows the main principle of MDA Model transformation:

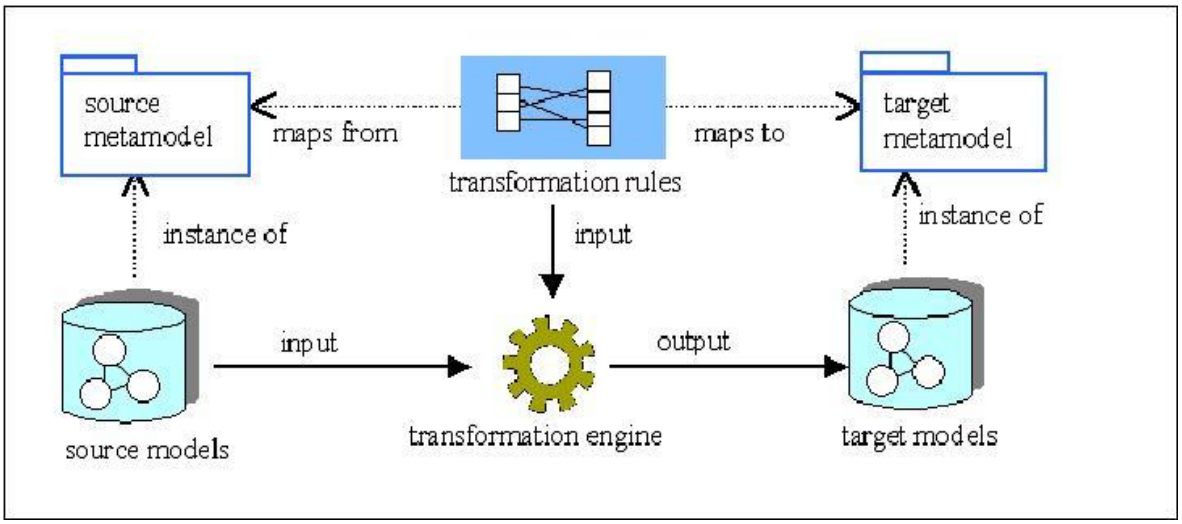


Fig 2.3 Model Transformation [Mod]

The transformation is between source and target models via a transformation engine, just like what is shown in the picture. For example, from a UML activity model into a UML statechart model. If we regard the transformation at a higher level, no matter what metamodels the source or target have, which models have "instance of" relationship with. And some transformation rules map both source and target metamodels to define how the transformation should work between different metamodels.

In this way, once metamodels, transformation rules and a transformation engine are ready, a kind of problems, instead of some certain problem, can be solved by the transformation.

2.2 OCL

The Object Constraint Language (OCL) is a notational language for analysis and design of software systems. It is part of the industry standard Unified Modeling Language (UML) that allows software developers to write constraints and queries over object models. [WK03] These constraints are particularly useful, as they allow a developer to create a highly specific set of rules that govern the aspects of an individual object. As many software projects today require unique and complex rules that are written specifically for business models, OCL is becoming an integral facet of object development.

OCL is a language that can express additional and necessary information about the models and other artifacts used in object-oriented modeling, and should be used in conjunction with UML diagrammatic models. Armed with OCL, UML can contain much more

information than it does alone. However, many OCL expressions cannot be presented in a UML diagram format.

OCL is a declarative, side-effects-free language; that is, the state of a system does not change because of an OCL expression. More importantly, a modeler can specify in OCL exactly what is meant, without restricting the implementation of the system that is being modeled. This enables a UML/OCL model to be completely platform-independent.

The OCL expression:

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

constrains the UML model example in Fig 2.4, it means in each flight, the passenger number cannot exceed the plane's seats number.

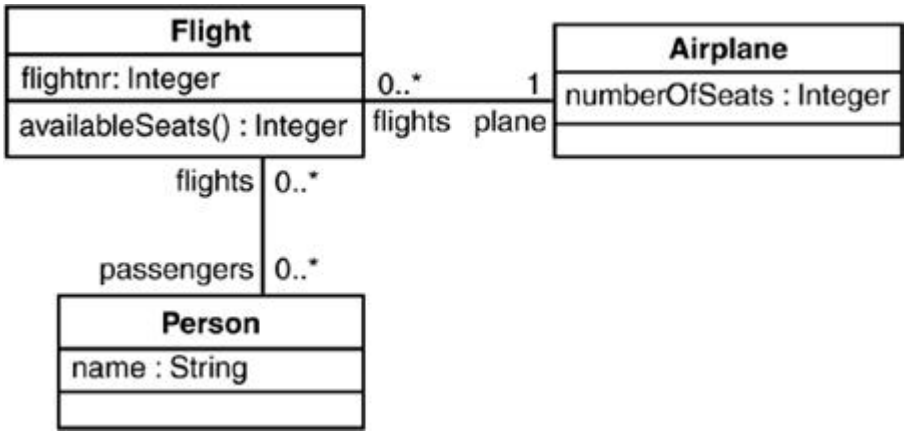


Fig 2.4 OCL constraint expression [WK03]

In UML 2, OCL can be used to write not only constraints, but also be used anywhere in the model to indicate a value. A value can be a simple value, such as an integer or a boolean, but it may also be a reference to an object, a collection of values, or a collection of references to objects. An OCL expression can represent, for example, a boolean value used as a condition in a statechart, or an association in class diagram. An OCL expression can be used to refer to a specific object in an interaction or object diagram. The next expression, for example, defines the body of the operation availableSeats() of the class Flight in Fig 2.4:

```
context Flight::availableSeats() : Integer
body: plane.numberOfSeats - passengers->size()
```

From the examples above, we can realize that OCL is a precise, unambiguous language that is easy for people who are not mathematicians or computer scientists to understand. It doesn't use any mathematical symbols, while maintaining mathematical rigor in its definition. OCL is a typed language, because it must be possible to check an OCL expression included in a specification without having to produce an executable version of the model.

2.3 Octopus (OCL Tool for Precise UML Specifications)

Octopus is a tool developed by Klasse Objecten, **http://www.klasse.nl/**. It offers two important functionalities:

1. Octopus is able to statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.

2. Octopus is able to transform the UML model, including the OCL expressions, into Java code. A GUI program following the MVC (Model-View-Controller) pattern may also be generated, with the Model being instances of classes for which Java code was also generated to check the OCL constraints.

Octopus has its own text based repredentation of UML. For example, the class Flight in Fig2.4 can be modeled in Octopus as (associations are omitted):

```
<class> Flight
<attributes>
+ flightnr: Integer;
<operations>
+ availableSeats(): Integer;
<endclass>
```

This text based .uml file in Octopus can be called Octopus model. In this thesis, Octopus was used in model transformation as an engine.

3 Transformation from state chart to Java

3.1 Analysis and Design

As explained in last chapter, to carry out a model transformation, we need to have the source and target metamodels and the transformation rule. In this section these three factors will be explained in detail.

3.1.1 Source: Statechart metamodel

From section 2.1.2 we know that the relationship between source model and source metamodel is “instance of”. In [KWB03], the authors describe metamodel as “language”, and the model and languages have an “is written in” relation, in fact, the principles are the same.

Since the statechart model will be the source model, to let the transformation work, a statechart metamodel or statechart language is needed.

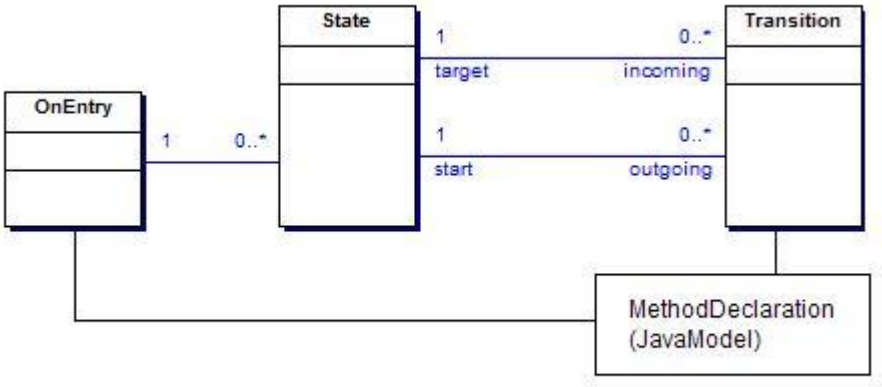


Fig 3.1 Statechart metamodel

Generally, in a UML statechart, there are 4 entries: State, Transition, Operation/Method and OnEntry. And the statechart language can be modeled as shown on Fig 3.1. Operation can

be modeled as MethodDelcaration, which is a part of Java metamodel and will be explained in next section. So the statechart metamodel (i.e. the definition of the statechart language) can be represented by the UML class diagram (Fig3.1). With this metamodel, any statecharts can be instantiated.

Octopus model

Because Octopus will work as transformation engine, the metamodel should be expressed in Octopus way. Octopus can represent a class diagram in its own text based way. For example, the *State* class and its associations with Transition class can be expressed as:

```
<endclass>
<class> State
<attributes>
+ name: String;
+ isInitial: Boolean;
<endclass>

<class> Transition<endclass>

<associations>
+ Transition.outgoing [0..*] <-> + State.start [1];
+ Transition.incoming [0..*] <-> + State.target [1];
```

For the full version of statechart metamodel in Octopus please check the corresponding section in Appendix.

Constraints

As a source model for MDA transformation, it should be as precise as possible. Therefore only the Octopus UML model above is not enough for the transformation using, some constraints and additional information should be combined with this metamodel together.

One state can have either an incoming Transition or outgoing Transition or both. But isolated states are not allowed. This constraint is defined with the following OCL.

```
context State
  inv notisolated:
   (not(self.incoming->isEmpty())) or
(not(self.outgoing->isEmpty()))
endpackage – statechart
```

Among all states, they must have different names, and there should be one and only one state can be initial state. The unique name and initial state constraints are expressed with OCL:

```
context State
  inv uniquename :
   self.allInstances()->isUnique(s: State| s.name)

context State
  inv onlyOneInitial:
    self.allInstances()->one(isInitial )
```

3.1.2 Target: Java metamodel

The goal of this MDA transformation is to get Java codes from a statecharts. Therefore, a Java metamodel should act as the target model. The Java metamodel should contain all entities in Java syntax. For instance, package declaration, type declaration, statements, method declaration and method body and so on should be included.

The Eclipse JDT provides APIs to manipulate Java source code, detect errors, perform compilations, and launch programs. Eclipse's JDT has its own Document Object Model (DOM) in the same spirit of the well-known XML DOM: the Abstract Syntax Tree (AST). It defines all java entities surrounding a CompilationUnit class and is helpful for constructing the Java metamodel in Octopus.

A part of the whole Java metamodel is seen in Fig 3.2. Even though it is not the complete model it has contained crucial Java elements to build a compilable Java compilation unit. With the Java metamodel any Java codes can be generated out. As same as the source model, the Java metamodel needs represented in Octopus model, which is the text based uml file. Further uses of this metamodel include custom refactorings, or program transformation in general.

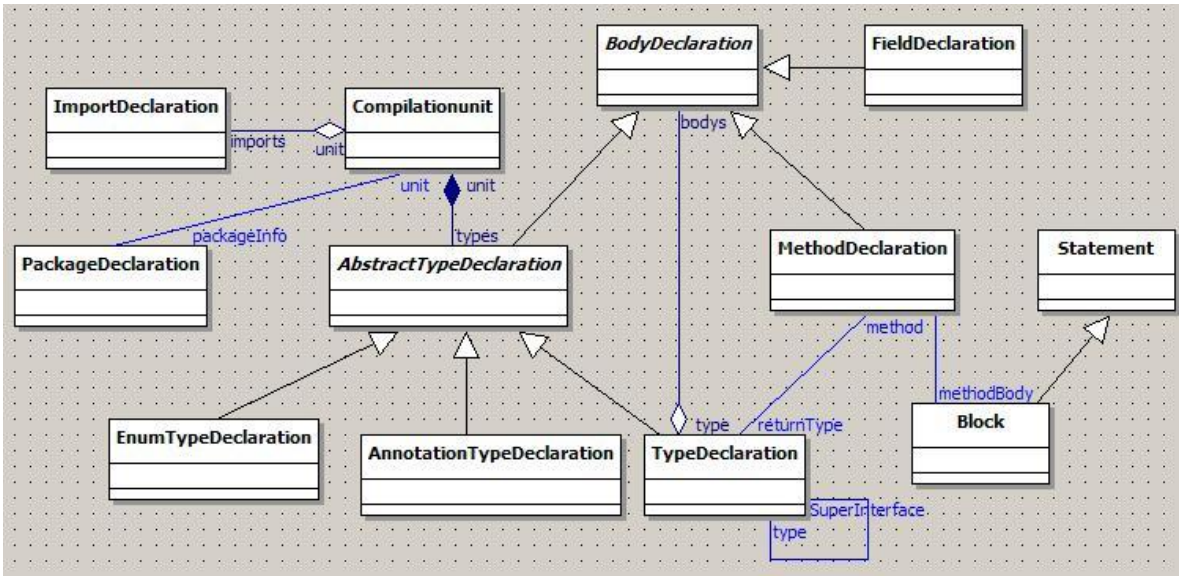


Fig 3.2 Java metamodel

Octopus model

The complete .uml file (javametamodel.uml) is available in Appendix. Here only codes related to a most common element in Java codes – Class (Type) are explained as example.

A Java class generally contains the class declaration, for example:

```
public class HelloWorld extends … {…}
```

and a class body. Typically, a class body contains some methods and fields. For example:

```
public class HelloWorld  {
  //class body – Fields :
  private String firstName;
  private String lastName;

  //class body – methods :
  public String sayHello{
   System.out.println("Hello, " +lastName);
  }
}
```

With the help of Fig 3.2, it is not difficult to understand how a Java class is modeled in this metamodel. A class is modeled as a TypeDeclaration, and it has a set of ordered components of BodyDeclaration, which is super type of FieldDeclaration and MethodDeclaration. The snippet of octopus model about Java class looks as follows:

```
+ <class> TypeDeclaration <specializes> AbstractTypeDeclaration
      <attributes>
      + isInterface :Boolean;
      <operations>
      + toJavaSource(): String;
      + getFields():OrderedSet(FieldDeclaration);
      + getMethods():OrderedSet(MethodDeclaration);
   <endclass>

+ <class> MethodDeclaration <specializes> BodyDeclaration
   <attributes>
   + isConstructor : Boolean;
   <operations>
   + isVarargs():Boolean;
   + parameters():OrderedSet(SingleVariableDeclaration);
   + toJavaSource():String;
   <endclass>

 + <class> FieldDeclaration <specializes> BodyDeclaration
      <operations>
       + toJavaSource():String;
      + fragments():OrderedSet(VariableDeclarationFragment);
   <endclass>

<associations>

--BodyDeclaration is a supertype, it could be fielddeclaration
or methoddeclaration--
+ TypeDeclaration.type [1] <aggregate>  ->
BodyDeclaration.bodyDeclarations [0..*]  <ordered>;
```

There is method called `toJavaSource()` in each class. This method is for code generation, it will be explained in detail in code generation section.

Constraints

Since the Java metamodel contains much more classes and much more complex logic than statechart metamodel, it has more constraints and OCL queries. In this section, only OCL constraints and queries concerning TypeDeclaration will be discussed as example.

There is an attribute `isInterface` defined with `boolean` type in class `TypeDeclaration`. This attribute distinguishes that if the class is an interface or not. By default, it should be initialized as false. The following OCL codes do that.

```
context TypeDeclaration::isInterface:Boolean
   init: false
```

Also for the method `getFields()` declared in `TypeDeclaration` class, it should get a list of fields declared in the class. It cannot be modeled in every detail neither with UML class diagram nor Octopus model. However, an OCL query can handle with it easily. The snippet below shows the query of method `getFields()` and `getMethods()`. In fact the OCL query has defined the rule of the method implementation. From MDA transformation perspective, this is rather helpful because it directly decides whether the final generated codes are precise and runnable.

```
context
TypeDeclaration::getFields():OrderedSet(FieldDeclaration)
   body:
      self.bodyDeclarations->
   select(oclIsTypeOf(FieldDeclaration))
      ->iterate(
      bodyD:BodyDeclaration;
fieldSet:OrderedSet(FieldDeclaration) |
         fieldSet->append(bodyD.oclAsType(FieldDeclaration))
      )

context
TypeDeclaration::getMethods():OrderedSet(MethodDeclaration)
   body:
      self.bodyDeclarations->
   select(oclIsTypeOf(MethodDeclaration))
      ->iterate(
         bodyD:BodyDeclaration;
methodSet:OrderedSet(MethodDeclaration) |
   methodSet->append(bodyD.oclAsType(MethodDeclaration))
      )
```

3.1.3 Transformation rule

After having source and target models in our left and right hands, we need something in between to let the transformation work. This is transformation rule. This section will talk about how to build the bridge between source and target models and connect them together.

Fig 3.1 explained the Statechart metamodel. And a MethodDeclaration class is associated with OnEntry and Transition classes in that diagram. This MethodDeclaration class is one element of Java metamodel. In statechart, one transition and onEntry can lead to an operation execution and this operation will map a Java method in the final java model. That is why the MethodDeclaration was introduced into statechart metamodel.

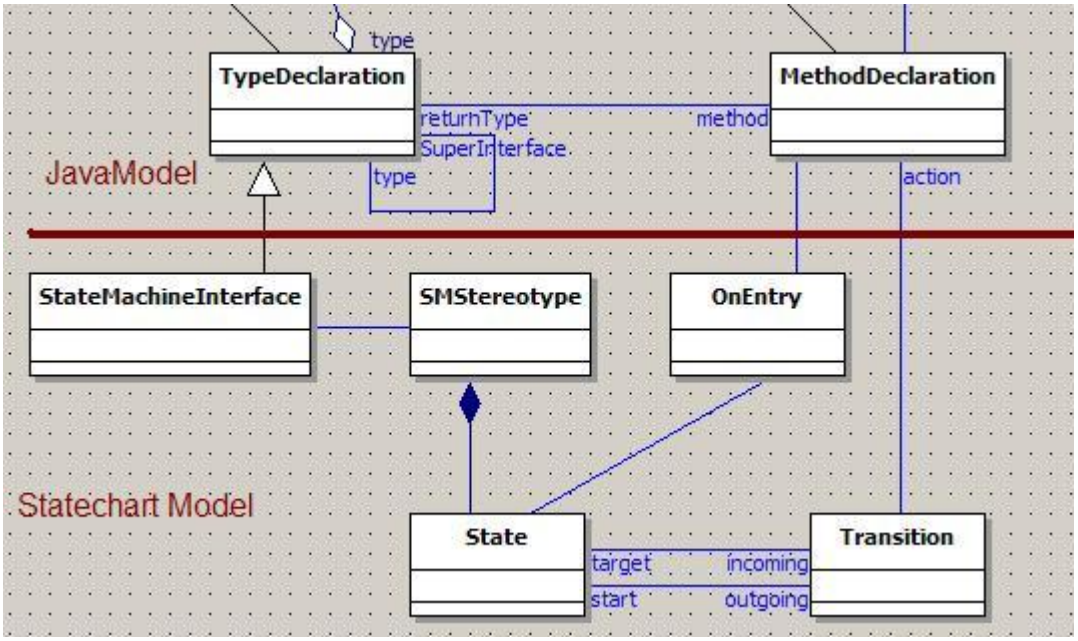


Fig 3.3 Transformation rule

Besides the connection to MethodDeclareation class, a new class which is called StaeMachineInterface is introduced into the picture. This class is a subclass of TypeDeclaration in Java model. As we have seen, some Java methods will be associated to OnEntries and Transitions to make sure all methods used by statechart model are subject to the well-formedness checks expressed in the metamodel (with OCL), any Java method used by statechart should be declared in this StateMachineInterface. And later this interface is implemented by the main business Java class. This interface will be explained in detail in section “Microwave oven example”.

There are some constraints on the StateMachineInterface. For example, it should always be

a Java interface instead of a Java class. Moreover, from statechart model needs, the methods declared in this interface should have no parameter.

The following OCL codes represent the constraints described above.

```
context StateMachineInterface
inv : isInterface
inv allMethodsParameterless :
getMethods().parameters()->size()=0
```

Since the final goal is to get Java codes from a given statechart, next section will elaborate how Java codes are generated automatically.

3.2 Java code generation

In Chapter 2, background, the two transformations for MDA have been introduced. The first one is from source model to target model. And the second one is what will be explained in this section, transformation from target model to specific source codes.

In this case, the target model is Java model. OCL query can be used for transformation from Java model to Java source codes. As known, source and target models are modeled in Octopus. Octopus has the feature that automatically generates the Java codes based on given models and OCL constrains. In fact the codes generated by Octopus are the infrastructure of MDA. During the generation, OCL query will be transformed into Java method implementation and finally the query result will be the Java method return value. Therefore an operation called `toJavaSource()` with return type `String` was added into each element needs generate codes of Java model. Simultaneously, OCL queries are created for each operation. Thus, in MDA infrastructure generated by Octopus, once the method `toJavaSource()`is called by an element of Java model, the Java codes will be generated out and return as a String.

To getting more concrete feeling about code generation, an example will be analyzed in this section to show how to generate a Java class with methods and fields.

A Java class is modeled as TypeDeclaration in Octopus Java model. Followings are the Octopus model about class:

```
+ <class> TypeDeclaration <specializes> AbstractTypeDeclaration
      <attributes>
      + isInterface :Boolean;
      <operations>
      + toJavaSource(): String;
      + getFields():OrderedSet(FieldDeclaration);
      + getMethods():OrderedSet(MethodDeclaration);
   <endclass>
```

A `TypeDeclaration` can be an interface, and also can have super class, implement other interface and a class can have different visibility properties, e.g. `public, protected` and so on. All those possibilities should be considered in the OCL query of operation `toJavaSource()` . Because Method and Field have their own `toJavaSource()` method and OCL queries, in the OCL query following, to generate the codes of methods and fields in the class, just simply calling the `toJavaSource()` method. Followings are the complete OCL query of `toJavaSource()` of `TypeDeclaration.` And it will return Java class with methods and fields.

```
context TypeDeclaration::toJavaSource():String
   body:
   let
   javaString:String = '',
   rear :String = '} \n',
   superinterfaces:OrderedSet(Type) = self.superInterfaceTypes

   in
      javaString.concat(
      if not self.annotation->isEmpty()
      then
         self.annotation.toJavaSource()
      else
         ''
      endif
      .concat('\n')).concat(
         self.modifiers->asOrderedSet()
            ->iterate( modifier:Modifier;
                     name :String = ' '|
                     name.concat( modifier.name.concat(' ') )
            ) -- iterator end
      )
            -- check isInterface
```

```
         .concat(
            if isInterface
            then -- $$$ interface
               ' interface
'.concat(self.typeName.identifier).concat(

                  if superinterfaces->size() > 0 -- has super
interfaces
                  then
                     ' extends
'.concat(superinterfaces->first().toJavaSource())
                     .concat(' { \n')
                  else
                     ' { \n '
                  endif

               )
               .concat(

   getMethods()->iterate( method:MethodDeclaration;
result:String = ''|

   result.concat(method.toJavaSource())  )--iterator end
                )

            else -- $$$ class
               ' class
'.concat(self.typeName.identifier).concat(
               if superinterfaces->size() > 0 -- has super
interfaces
               then

               ' implements '.concat(
                  superinterfaces->iterate(
                     interfaceType:Type;
                     typeName:String = ''|

   typeName.concat(interfaceType.toJavaSource())
                     .concat(
                        if
(superinterfaces->indexOf(interfaceType)
```

```
                           = superinterfaces->size())
                        then
                           ' { \n'
                        else
                           ', '
                        endif
                     ) -- now public... class Cname implements
a,b,c {
                  ) -- iterator superinterfaces end
               )
               else
                  -- here "extends" could be added in future
within another if cause

                  '-- testing String here --'
               endif

                  )
               .concat(

   getMethods()->iterate( method:MethodDeclaration;
result:String = ''|
                  result.concat(method.toJavaSource()))
                )
            endif

      ).concat(rear)
```

3.3 Example: “Microwave oven”

So far in this chapter, how to transform from statechart to Java source codes has been analyzed. This section will show an example to present how to work with this infrastructure.

3.3.1 Microwave oven statechart

In this example, a microwave oven Java package should be generated from the given statechart. First of all let us take a look the statechart of microwave oven (Fig 3.4).

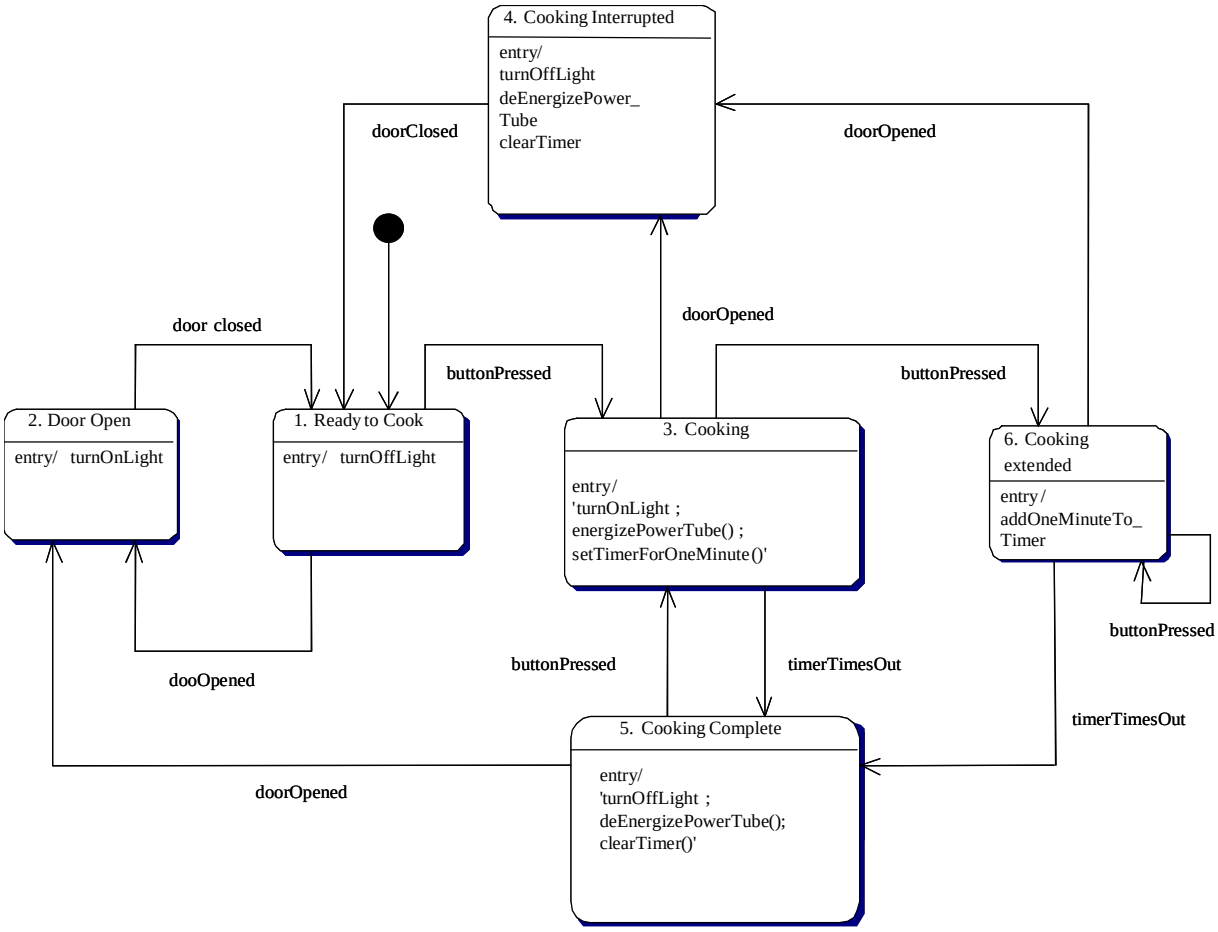


Fig 3.4 Microwave oven statechart

The above statechart shows the all states switch of a microwave oven. To make the example simpler and to show how to generate Java codes with the MDA infrastructure more clearly, only two states with their transitions will be generated. The two states are shown in Fig 3.5, they contain states, transitions and operations.

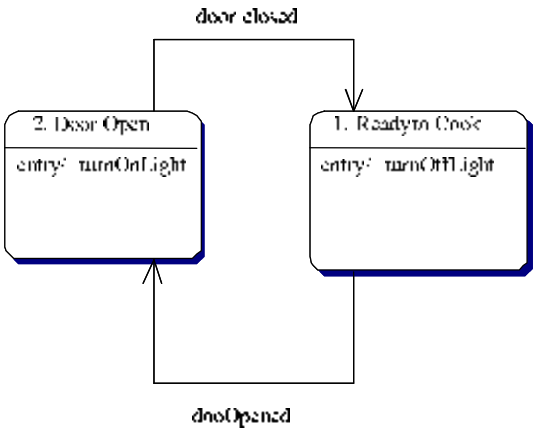


Fig 3.5 simplified microwave oven statechart

3.3.2 The final generated Java codes

The final Java source codes will be compatible with SUN JDK 1.5. There is a new feature in JDK1.5, annotation. This feature also used by the microwave oven example for further generation of AOP (Aspect-oriented programming) code.

In general, the Java classes of microwave oven example contain

1. `IMicrowave.java`, which defines the methods used by states' transitions
2. `Microwave.java`, the main business class
3. `OutgoingTransitions.java`, defines the outgoingtransition annotation
4. `States.java`, which is an enumeration type contains all states
5. `StateSpace.java`, which defines the annotation for states
6. `Transition.java`, defines the annotation for transitions

In the final microwave codes, some classes and annotations are not needed to be generated by MDA. They are predefined annotations `StateSpace.java, Transition.java` and `OutgoingTransitions.java`. Followings are the annotation declarations.

`StateSpace.java`

```
package generated;
import java.lang.annotation.*;
```

```
@Documented
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface StateSpace {
   String[] states() ;
   String initial() ;
}
```

OutgoingTransitions.java.

```
package generated;
import java.lang.annotation.*;

@Documented
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface OutgoingTransitions {
   Transition[] value() ;
}
```

Transition.java

```
package generated;

import java.lang.annotation.*;

@Documented
@Target({ElementType.METHOD, ElementType.TYPE,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Transition {
   States from() ;
   States to() ;
}
```

Besides the above annotation declaration files, all Java classes should be generated automatically. Next the left three classes will be explained one by one.

States.java

`States.java` is an enumeration type, which is a new feature of Sun JDK 1.5 as well. Any available states should be contained by this enumeration. And it is not possible for user to use any state which is not defined in it. The generated codes of States.java should look like:

```
package generated;

public  enum States {
   DOOROPEN,
   READYTOCOOK,
   COOKING,
   COOKINGINTERRUPTED,
   COOKINGCOMPLETE,
   COOKINGEXTENDED;
}
```

IMicrowave.java

`IMicrowave.java` is an interface that contains all operations used by transitions and onEntries. This interface will be implemented by the main business class `Microwave.java`, and those methods declared in it will be implemented in `Microwave` class.

The generated codes of IMicrowave.java interface should look like

```
package generated;

@StateSpace (
   states =
{"DOOROPEN","READYTOCOOK","COOKING","COOKINGINTERRUPTED","COO
KINGCOMPLETE","COOKINGEXTENDED"},
initial = "READYTOCOOK"
)

 public  interface IMicrowave {
   public  void openDoor () ;
   public  void closeDoor () ;
   //... other methods here.
}
```

Microwave.java

Microwave is the main business class, as described in `IMicrowave` interface, this class will implement `IMicrowave` interface.

```
package generated;

public  class Microwave implements IMicrowave {
```

And it should have a constructor to instantiate a Microwave oven object.

```
public Microwave () {
   System.out.println("constructor");
  }
```

Then all methods defined in the `IMicrowave` interface should be implemented in the class. Also all state switch related to certain method should be represented in `annotation` of the method. For example, the `openDoor()` and `closeDoor()` methods and their annotations should look like following snippet. The state switch of each method is based on the statechart Fig 3.4.

```
@OutgoingTransitions (
   {@Transition (
   from = States.READYTOCOOK, to = States.DOOROPEN ),

   @Transition (
   from = States.COOKING,
   to = States.COOKINGINTERRUPTED
   ),

   @Transition (
   from = States.COOKINGCOMPLETE, to = States.DOOROPEN ),

   @Transition (
   from = States.COOKINGEXTENDED,
   to = States.COOKINGINTERRUPTED
```

```
   )
   }
)
 public  void openDoor () {
// here is the method body
}
@OutgoingTransitions (
{
  @Transition (
  from = States.DOOROPEN,to = States.READYTOCOOK ),

  @Transition (
  from = States.COOKINGINTERRUPTED, to = States.READYTOCOOK )
})
 public  void closeDoor () {
// here is the method body
}
}
```

Generating the classes above for the Microwave oven example comprises 3 steps:

1, generate the MDA infrastructure based on statechart model, Java model and their OCL constraints with Octopus.

2, create a main class, instantiate all state objects, transition objects, method instances and so on. For example to instantiate `openDoor()` method:

```
// #create methods public void openDoor()
      MethodDeclaration openDoor = new MethodDeclaration();
      Type returnType = new PrimitiveType();
      ArrayList modifiers = new ArrayList();
      modifiers.add(this.PUBLIC);
      MethodDeclaration iopenDoor =
getMethodDeclaration(modifiers,"openDoor", this.VOID);
```

To create states

```
// $$$ create states and add them into sms
      State s_DOOROPEN = new State();
      s_DOOROPEN.setName("DOOROPEN");
      State s_READYTOCOOK = new State();
```

```
      s_READYTOCOOK.setName("READYTOCOOK");
      s_READYTOCOOK.setInitial(true);
      State s_COOKING = new State();
      s_COOKING.setName("COOKING");
      State s_COOKINGINTERRUPTED = new State();
      s_COOKINGINTERRUPTED.setName("COOKINGINTERRUPTED");
      State s_COOKINGCOMPLETE = new State();
      s_COOKINGCOMPLETE.setName("COOKINGCOMPLETE");
      State s_COOKINGEXTENDED = new State();
      s_COOKINGEXTENDED.setName("COOKINGEXTENDED");
```

Create State machine stereotype and add states in:

```
      // $$$ create SMStereotype ins -> sms
      SMStereotype sms = new SMStereotype();
      // $$$ ADD STATES and interface INTO SMS
      sms.addToStates(s_DOOROPEN);
      sms.addToStates(s_READYTOCOOK);
      sms.addToStates(s_COOKING);
      sms.addToStates(s_COOKINGINTERRUPTED);
      sms.addToStates(s_COOKINGCOMPLETE);
      sms.addToStates(s_COOKINGEXTENDED);
```

For full version of the main class of microwave oven please check appendix corresponding section.

3, Call `toJavaSource()` method of each class instance to get the final Java source codes.

Since last section has explain the mechanism of generating Java codes, here it is very easy to get the final codes once have `CompilationUnit` instance of `Microwave` class, `IMicrowave` class and `State` class. The following snippet is the codes in main class to generate Java codes. `statesUnit`, `iMicrowaveUnit` and `microwaveCLSUnit` in the codes are instances of `CompilationUnit`.

```
this.write("src\\generated\\States.java",
statesUnit.toJavaSource());
this.write("src\\generated\\IMicrowave.java",
iMicrowaveUnit.toJavaSource());
```

```
this.write("src\\generated\\Microwave.java",
microwaveCLSUnit.toJavaSource());
```

4 MDA in enterprise architecture

An infrastructure of MDA based on statechart and Java source codes were explained in Chapter 3. In fact, MDA can help transformation between any different software artifacts. Enterprise applications, especially web based enterprise applications, are widely used today. In an enterprise application, many different software artifacts are usually involved. For example, a J2EE application, generally, there should be a deployment descriptor for all enterprise java bean configuration. The EJBs and deployment descriptors are different artifacts, and usually we should manually check that they agree with each other. One goal of this thesis is to establish an MDA infrastructure target on enterprise architecture, to make the transformation between different software artifacts automatic.

In this thesis, Web service based on JBoss was chosen as the target enterprise architecture. This chapter will elaborate how to build an MDA infrastructure to automatically transform from SecureUML model to JBoss web service authorization configuration.

4.1 SecureUML

Security is an important aspect, influencing application design and development. SecureUML [LBD] is a modeling language for modeling access control policies and their integration into a model-driven software development process.

Role-Based Access Control is the foundation of SecureUML. Fig 4.1 describes the data model of RBAC.

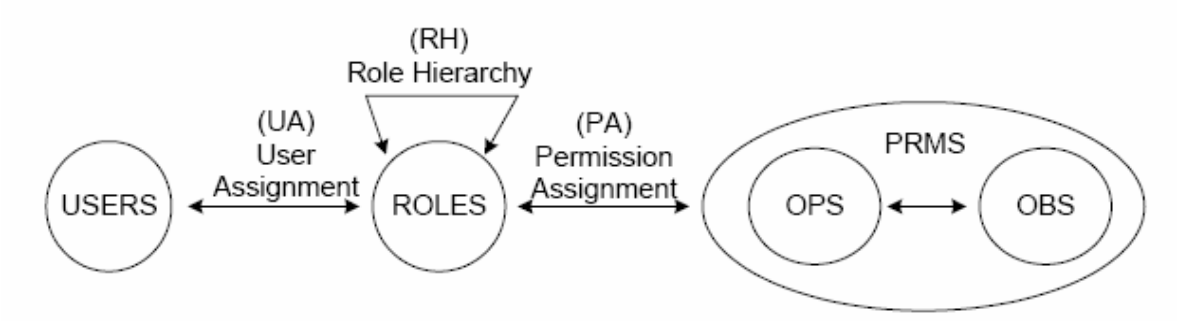


Fig 4.1 Role-Based Access Control [LBD]

The model consists of five data types: users (USERS), roles (ROLES), objects (OBS), operations (OPS) and permissions (PRMS). A user is defined as a person or a software agent. A role is a job or function within an organization. It combines all privileges needed to fulfill the respective job or function. Privileges are expressed in terms of the permissions assigned to a role by entries to the relation Permission Assignment. A permission represents the authorization to execute an operation on one or more protected objects or resources. An object in this context is a system resource or a set of resources that are protected by the security mechanism. An operation is an action on a protected object that can be initiated by a system entity. The types of operations depend on the type of the protected objects. In a file system, for example, there might be permissions to read, write or execute files. The assignment of roles to users is defined by the relation User Assignment. The relation Role Hierarchy defines an inheritance relationship between roles. A relation r1 inherits r2 implies that all permissions of role r2 are also permissions of role r1. [LBD]

SecureUML is based on an extended model for role-based access control (RBAC). RBAC is a well-established access control model with widely-recognized advantages and it is supported by a large number of software platforms. However, generally RBAC lacks support for expressing access control conditions that refer to the state of a system, for example, the state of a protected resource, parameter values, date or time. [LBD] Therefore, the concept of authorization constraints was introduced. An authorization constraint is a precondition for granting access to an operation. And OCL is the language to define the authorization constraint.

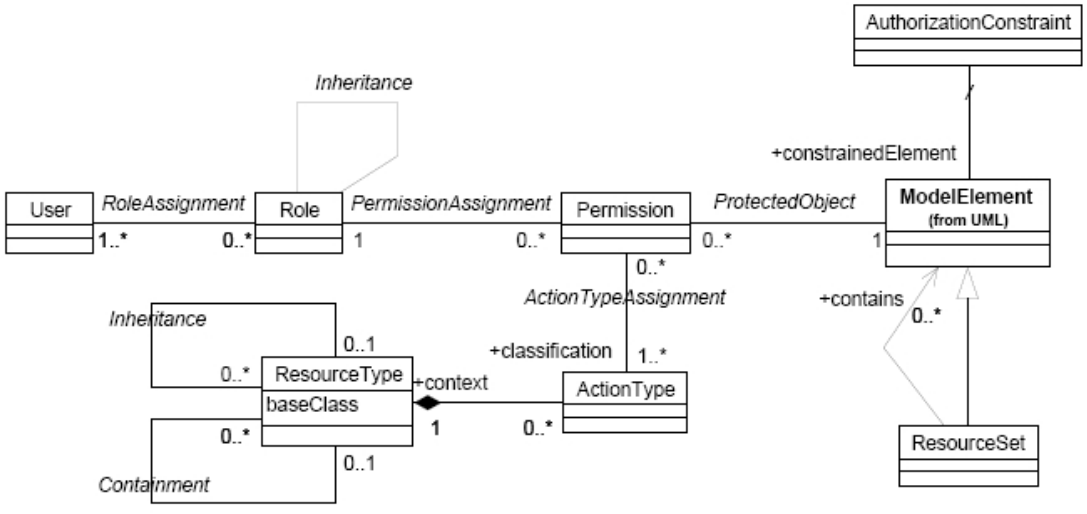


Fig 4.2 SecureUML Metamodel [LBD]

The SecureUML metamodel, shown in Figure 4.2, is defined as an extension of the UML metamodel. The entities of RBAC are represented directly as metamodel types. The metamodel types User, Role and Permission as well as relations between these types are used. In order to smoothly integrate SecureUML into other modeling languages, protected resources are represented as follows: instead of defining a dedicated metamodel type to represent them, every UML model element is allowed to take the role of a protected resource. In addition, the ResourceSet type is introduced which represents a user-defined set of model elements for defining permissions or authorization constraints.

A permission is a relation object connecting a role to a ModelElement or a ResourceSet. The semantics of a permission are defined by the ActionType elements used to classify the permission (refer to the association ActionTypeAssignment).[IO] Every action type represents a class of security-relevant operations on a particular type of protected resource. For example, the action type execute is obvious for the business method resource type, whereas the action types read and change are normally used for the business attribute resource type.

4.2 Web service authorization in JBoss

Web service is playing more and more important role in enterprise application integration and B2B applications. And the common way to build a web service is exposing existing components as service endpoints, for example, Stateless session beans (SLSB) and sevlets can be exposed as web services.

After components were exposed as web service, the security issues should be addressed. For example, how to make sure that a service caller has the certain right to access requested resource or execute some operations. Usually, different application servers have their own dialect in the deployment descriptor to define the security rules. In this section, it is explained how to secure a web service endpoint exposed from a SLSB in JBoss.

Assume that we have a SLSB `ToDoListEntryBean`, which has methods `update`, `read`, `create` and `delete` todolist entries, with authorization depending on the invoker's role. In this example, there are three roles, `admin`, `owner` and `user`. `Admin` can execute any methods declared by the bean, `owner` have the right to execute `update`, `read` and `create` methods but cannot `delete` todolist entries. `User`, however, only `read` entry information from the bean. As explained above, this bean can be exposed as a service endpoint. To secure this web service endpoint in JBoss, there are two steps.

Step 1, Secure the access to the SLSB in normal way, which means that just like it is not a service endpoint.

In JBoss, this is done with `ejb-jar.xml`. In this configuration file, the method permissions for the SLSB endpoint are setup. Note that it is neither necessary nor required for the endpoint to have home/remote interfaces [JBo05].

The first part of the configuration is regarding the bean information, the configuration snippet following shows how the bean is defined.

```
 <enterprise-beans>
   <session>
     <ejb-name> ToDoListEntryBean </ejb-name>
     <service-endpoint>myExample.endPoint</service-endpoint>
     <ejb-class>mypackage.ToDoListEntryBean </ejb-class>
     <session-type>Stateless</session-type>
     <transaction-type>Container</transaction-type>
     <security-role-ref>
       <role-name>admin</role-name>
     </security-role-ref>
   </session>
 </enterprise-beans>
```

Besides the bean information, another configuration element is `<assembly-descriptor>`. It contains `<security-role>` and `<method-permission>` settting.

Since we have three different roles defined for accessing this bean, they are configured as follows:

```
<security-role>
   <role-name>admin</role-name>
   <role-name>owner</role-name>
   <role-name>user</role-name>
</security-role>
```

Each role defined in code fragment above is authorized to access certain methods against the rule described. The mapping between role and methods are defined as follows. `method-permission` setting:

```
<method-permission>
   <role-name>user</role-name>
   <method>
      <ejb-name>ToDoListEntryBean</ejb-name>
      <method-name>getEntryName</method-name>
      <method-name>getEntryValue</method-name>
   </method>
</method-permission>

<method-permission>
  <role-name>owner</role-name>
  <method>
     <ejb-name>ToDoListEntryBean</ejb-name>
     <method-name>getEntryName</method-name>
     <method-name>getEntryValue</method-name>
     <method-name>createEntry</method-name>
     <method-name>setEntryName</method-name>
     <method-name>setEntryValue</method-name>
  </method>
</method-permission>

<method-permission>
  <role-name>admin</role-name>
  <method>
     <ejb-name>ToDoListEntryBean</ejb-name>
     <method-name>deleteEntry</method-name>
     <method-name>getEntryName</method-name>
     <method-name>getEntryValue</method-name>
     <method-name>createEntry</method-name>
     <method-name>setEntryName</method-name>
     <method-name>setEntryValue</method-name>
  </method>
</method-permission>
```

Step 2, define the security domain for this deployment. The JBossWS security context is configured in `login-config.xml` and uses the `UsersRolesLoginModule` [JBo05].

```
<jboss>
  <security-domain>java:/jaas/JBossWS</security-domain>
  <enterprise-beans>
```

```
    <session>
      <ejb-name>ToDoListEntryBean</ejb-name>
      <jndi-name>ejb/ToDoListEntryBean</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

4.3 JBoss deployment descriptor generation

In section 4.1 we have seen that SecureUML allows a more understandable modeling of access control. And the web service endpoint security is mainly concerning the authorization issue of the endpoint. Our goal is to automatically generate from a given SecureUML model the JBoss deployment descriptor fragments, which target on securing a service endpoint.

This section will focus on how to build the MDA infrastructure to reach this goal.

4.3.1 SecureUML Octopus model

As same as chapter 3, the transform engine is based on Octopus as well. The first thing should do is create the octopus model of the source metamodel, in this case, the secureUML metamodel.

Section 4.1 has explained the SecureUML metamodel in detail. The main entities of SecureUML are `Role`, `Permission`, `ActionType`, `ResourceType` and `AuthorizationConstraint`. They can be modeled in Octopus as follows:

```
+<class> Role
   <attributes>
   + name:String;
   <operations>
   + toEjbDescriptor():String;
<endclass>

+<class> Permission
   <attributes>
   + name:String;
<endclass>

+<class> ActionType
```

```
   <attributes>
   + name:String;
<endclass>
```

```
+<class> ResourceSet
   <attributes>
   + name:String;
<endclass>

+<class> ResourceType
   <attributes>
   + name:String;
<endclass>

+<class> AuthorizationConstraint
   <attributes>
   + name:String;
<endclass>
```

In class `Role`, there is an operation `toEjbDescriptor()` that has not been mentioned during our discussion of SecureUML. This operation is for generating the role definition snippet of deployment descriptor, and it will be explained in detail in section 4.3.

The relations among those SecureUML entities can be represented as follows in Octopus model:

```
<associations>
+ Role.relatedRole [1..*] <->
Permission.permissionAssignment[0..*];

+ ActionType.actionTypeAssignment[1..*] <->
Permission.withPermission[0..*];

+ Permission.withPermission[0..*] <->
ResourceSet.protectObject[1..*];

+ ResourceType.classification[1] <aggregate> <->
ActionType.context[0..*];
```

The secureUML Octopus model can be represent in UML class diagram below:

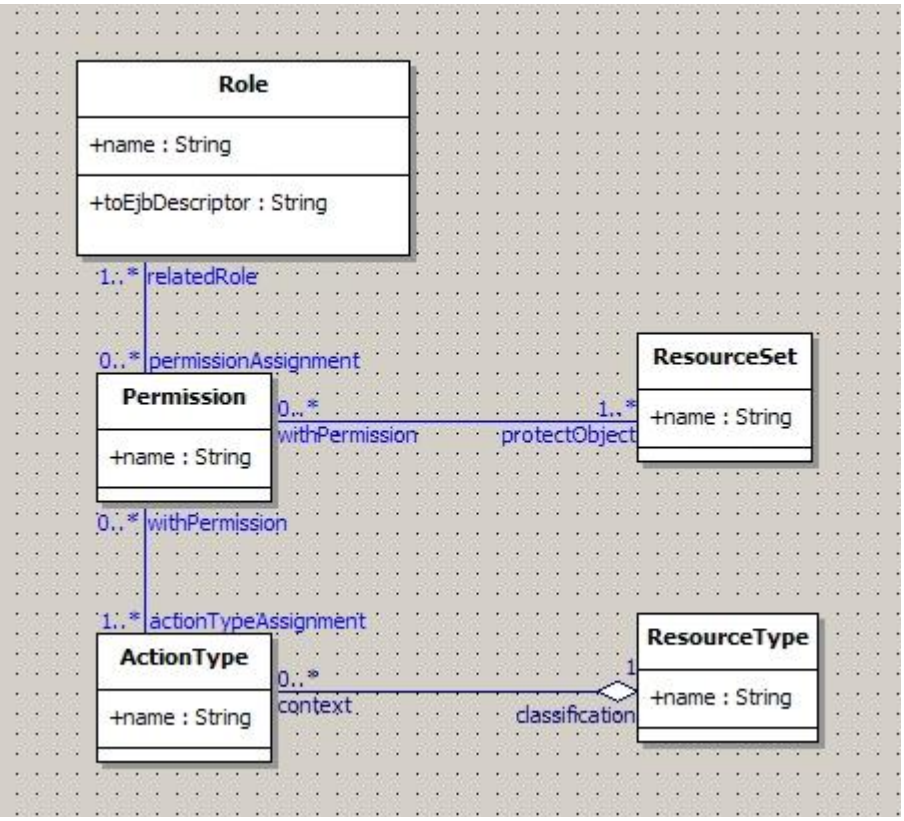


Fig 4.3 secureUML model class diagram

4.3.2 SLSB Octopus model

Section 4.2 described how to secure an EJB based web service endpoint. Besides SecureUML, EJB is another building block of the MDA infrastructure. Therefore the EJB should be modeled into Octopus model as well. However, since we target on the deployment descriptor, not all EJB information is required to be modeled. Considering the EJB configuration in deployment descriptor, only the bean, methods and attributes are necessary to be represented into Octopus model. The following Octopus model snippet shows how the three elements look like.

```
+<class> EjbAttribute
   <attributes>
   + name:String;
<endclass>

+<class> EjbMethod
```

```
   <attributes>
   + name:String;
   + actionFlag:String;
<endclass>

+<class> EjbClass
   <attributes>
   + name:String;
<endclass>
```

The `EjbClass` and `EjbAttribute` are easy to understand. There is a special attribute `actionFlag` in `EjbMethod` class. This attribute indicates the method type, for example, `update`, `read`, `create` or `delete`. The method types are the mapping of corresponding `ActionTypes` in SecureUML model. They help us to examine if a role has the right to execute the requesting method.

Obviously, an EJB can have many attributes and methods. This association is represented as follows in Octopus.

```
<associations>

+ EjbClass.ejbClass [1] <aggregate> ->
EjbAttribute.ejbAttribute[0..*];

+ EjbClass.ejbClass [1] <aggregate> ->
EjbMethod.ejbMethod[0..*];
```

The following figure shows the Ejb model in UML class diagram.

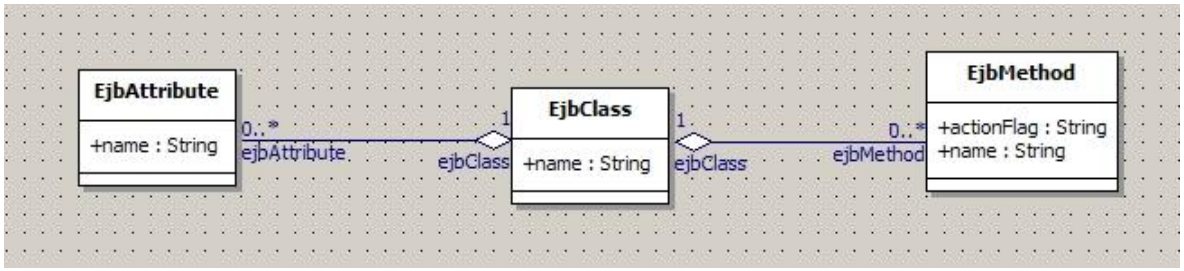


Fig 4.4 EJB model in UML class diagram

4.3.3 Deployment descriptor snippet generation

So far we have SecureUML and SLSB Octopus model in hand. Both of them are part of source model. A bridge is missing to connect them. So a class `WSEndpoint` is introduced into the picture. The whole picture of this MDA infrastructure has been drawn out as what the Fig 4.5 shows.

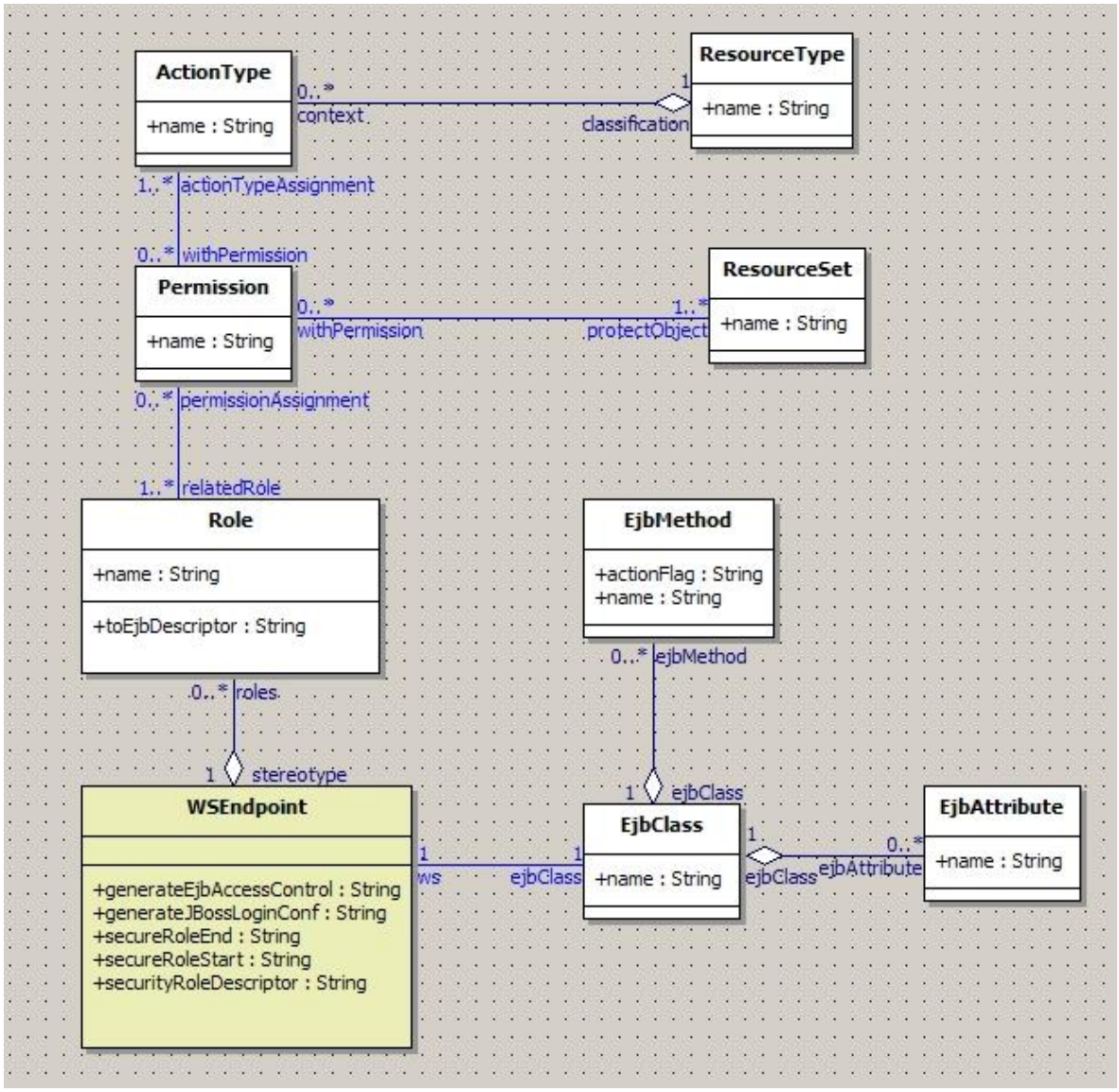


Fig 4.5 the UML diagram of WebService endpoint security

The class `WSEndpoint` can have any number of roles, and also have an `ejbClass` attribute. Following codes shows the `WSEndpoint` class Octopus model and associations between `WSEndpoint, Role` and `EjbClass` in Octopus.

```
+<class> WSEndpoint

<operations>
   + secureRoleStart():String;
   + secureRoleEnd():String;
   + securityRoleDescriptor():String;
   + generateEjbAccessControl():String;
   + generateJBossLoginConf() :String;
<endclass>

<associations>
+ WSEndpoint.stereotype [1] <aggregate> -> Role.roles[0..*];
+ WSEndpoint.ws[1] -> EjbClass.ejbClass[1];
```

The operations declared in `WSEndpoint` class are for code generation purpose. In this MDA infrastructure, code generation is still built based on OCL queries. Which means any operations of `WSEndpoint` class will have an OCL query corresponded. Next, OCL queries will be explained in detail.

From section 4.2, we know that there are a total of three parts which are relevant to web service endpoint authorization in the deployment descriptor.

1) In `ejb-jar.xml`, `<security-role>` should be defined. This configuration code can be generated by the OCL query of method `toEjbDescriptor`() of `Role` class. In section 4.2 this method was mentioned. In the definition, all role names should be listed in `<security-role>` element. The OCL query below outputs the role items.

```
context Role::toEjbDescriptor() : String
body: let roleString :String = '' in
roleString.concat('
<role-name>'.concat(self.name).concat('</role-name>\n'))
```

For the parent element `<security-role>`, it is a static element, so following simple OCL queries just directly output the start and end elements.

```
context WSEndpoint::secureRoleStart() : String
body: '<security-role>\n'

context WSEndpoint::secureRoleEnd() : String
body: '</security-role>\n'
```

2) Still in `ejb-jar.xml`, all methods permissions should be defined. They look like:

```
<method-permission>
     <role-name>rolename</role-name>
     <method>
       <ejb-name>ejbname</ejb-name>
       <method-name>methodname</method-name>
     </method>
</method-permission>
```

Operations `securityRoleDescriptor()` and `generateEjbAccessControl()` are responsible for generating this part of definition codes. Followings are the OCL queries of the two operations.

```
context WSEndpoint:: securityRoleDescriptor() :String
body:let roleString :String = '' in
roleString.concat(secureRoleStart())
.concat(
   roles->iterate( role:Role;
                  result:String = ' '|
                  result.concat( role.toEjbDescriptor() )
         )
   )
.concat(secureRoleStart())
```

The OCL query above outputs `<role-name>rolename</role-name>`. This query is called by the following OCL query.

```
context WSEndpoint:: generateEjbAccessControl() : String
body:
roles->iterate(role:Role; resultStr:String = ''|
      resultStr.concat('\n<method-permission>\n')
      .concat(role.toEjbDescriptor())
      .concat('<method>\n')
   .concat('<ejb-name>').concat(ejbClass.name).concat('</ejb-
name>\n')
      .concat( role.permissionAssignment -> iterate(
               permission :Permission; result1:String = '' |
               result1.concat(
                  permission.actionTypeAssignment -> iterate(
                        actionType:ActionType;
                        result2:String = '' |
```

```
                        result2.concat(
                           ejbClass.ejbMethod ->iterate(
                                 methd:EjbMethod;
                                 result3:String = ''|
                                 result3.concat(
                                 if methd.actionFlag =
actionType.name
                                 then
                                    '<method-name>'
              .concat(methd.name).concat('</method-name>\n')
                                 else
                                 ''
                                 )
                              )
                           )
                     )
               )
            )
   ).concat('</method>\n').concat('</method-permission>\n')
   )
```

3) The last step defines the login configuration in `login-config.xml`.

From code generation perspective, in this step, the only dynamic codes are the ejb name others are all static elements. Therefore the OCL query is rather simple.

```
context WSEndpoint:: generateJBossLoginConf() : String
body:
'<jboss>\n'
.concat('
<security-domain>java:/jaas/JBossWS</security-domain>\n')
.concat('     <enterprise-beans>\n')
.concat('        <session>\n')
.concat('
<ejb-name>').concat(ejbClass.name).concat('</ejb-name>\n')
.concat('
<jndi-name>ejb/').concat(ejbClass.name).concat('</jndi-name>\
n')
.concat('        </session>\n')
.concat('     </enterprise-beans>\n')
```

```
.concat('</jboss>\n')
```

After Octopus models and OCL queries are ready a main class is need to instantiate SecureUML metamodel and EJB metamodel. The main class will be analyzed next.

As defined before, the `ToDoListEntryBean` has four types of method: `read`, `update`, `create` and `delete`. In fact, the method type is the `ActionType` in secureUML model. Therefore create `ActionType` objects as following:

```
private final ActionType ejbRead = new ActionType("read");
private final ActionType ejbUpdate = new ActionType("update");
private final ActionType ejbCreate = new ActionType("create");
private final ActionType ejbDelete = new ActionType("delete");
```

We have known that class WSEndpoint is the bridge connecting SecureUML and EjbClass. So it should be instantiated first.

```
WSEndpoint  wsEndpoint = new WSEndpoint();
```

Then create the EjbClass object with given name, and the attributes and methods.

```
//initialize the EJB : toDoListEntry
EjbClass toDoListEntry = new EjbClass("ToDoListEntryBean");

// attributes
EjbAttribute entryName = new EjbAttribute("entryName");
EjbAttribute entryValue = new EjbAttribute("entryValue");

// methods
   EjbMethod entryNameSetter = new
EjbMethod("setEntryName","update");
   EjbMethod entryNameGetter = new
EjbMethod("getEntryName","read");
   EjbMethod entryValueSetter = new
EjbMethod("setEntryValue","update");
   EjbMethod entryValueGetter = new
EjbMethod("getEntryValue","read");
   EjbMethod entryCreate = new
EjbMethod("createEntry","create");
   EjbMethod entryDelete = new
EjbMethod("deleteEntry","delete");
```

```
   toDoListEntry.addToEjbAttribute(entryName);
   toDoListEntry.addToEjbAttribute(entryValue);
   toDoListEntry.addToEjbMethod(entryNameSetter);
   toDoListEntry.addToEjbMethod(entryNameGetter);
   toDoListEntry.addToEjbMethod(entryValueSetter);
   toDoListEntry.addToEjbMethod(entryValueGetter);
   toDoListEntry.addToEjbMethod(entryCreate);
   toDoListEntry.addToEjbMethod(entryDelete);
```

So far, the ToDoListEntryBean with its attributes and methods have been instantiated. Then all entities in secureUML model should be instantiated and initialized.

```
//roles
Role user = new Role("user"); // read only
Role owner = new Role("owner"); // read, update,create
Role admin = new Role("admin");  // read,update,create,delete

//Permission
Permission userPermission = new Permission("userPermission");
Permission ownerPermission = new Permission("ownerPermission");
Permission adminPermission = new Permission("adminPermission");

// assign actiontype and permission
userPermission.addToActionTypeAssignment(ejbRead);


   ownerPermission.addToActionTypeAssignment(userPermission.g
etActionTypeAssignment());
ownerPermission.addToActionTypeAssignment(ejbUpdate);
ownerPermission.addToActionTypeAssignment(ejbCreate);
adminPermission.addToActionTypeAssignment(ownerPermission.get
ActionTypeAssignment());
adminPermission.addToActionTypeAssignment(ejbDelete);

      // assign permission to roles
user.addToPermissionAssignment(userPermission);
owner.addToPermissionAssignment(ownerPermission);
admin.addToPermissionAssignment(adminPermission);
```

Finally, against the associations between `WSEndpoint`, `Role` and `EjbClass`, assign the

`WSEndpoint` object as following:

```
      wsEndpoint.addToRoles(user);
      wsEndpoint.addToRoles(owner);
      wsEndpoint.addToRoles(admin);
      wsEndpoint.setEjbClass(toDoListEntry);
```

Now invoking the certain method of `wsEndpoint` object, the predefined OCL query will generate the corresponding security definition. For example, to generate the security-role configuration in `ejb-jar.xml` deployment descriptor, we just call

```
wsEndpoint.securityRoleDescriptor();
```

To generate Ejb method access permission configuration, we can call

```
wsEndpoint.generateEjbAccessControl();
```

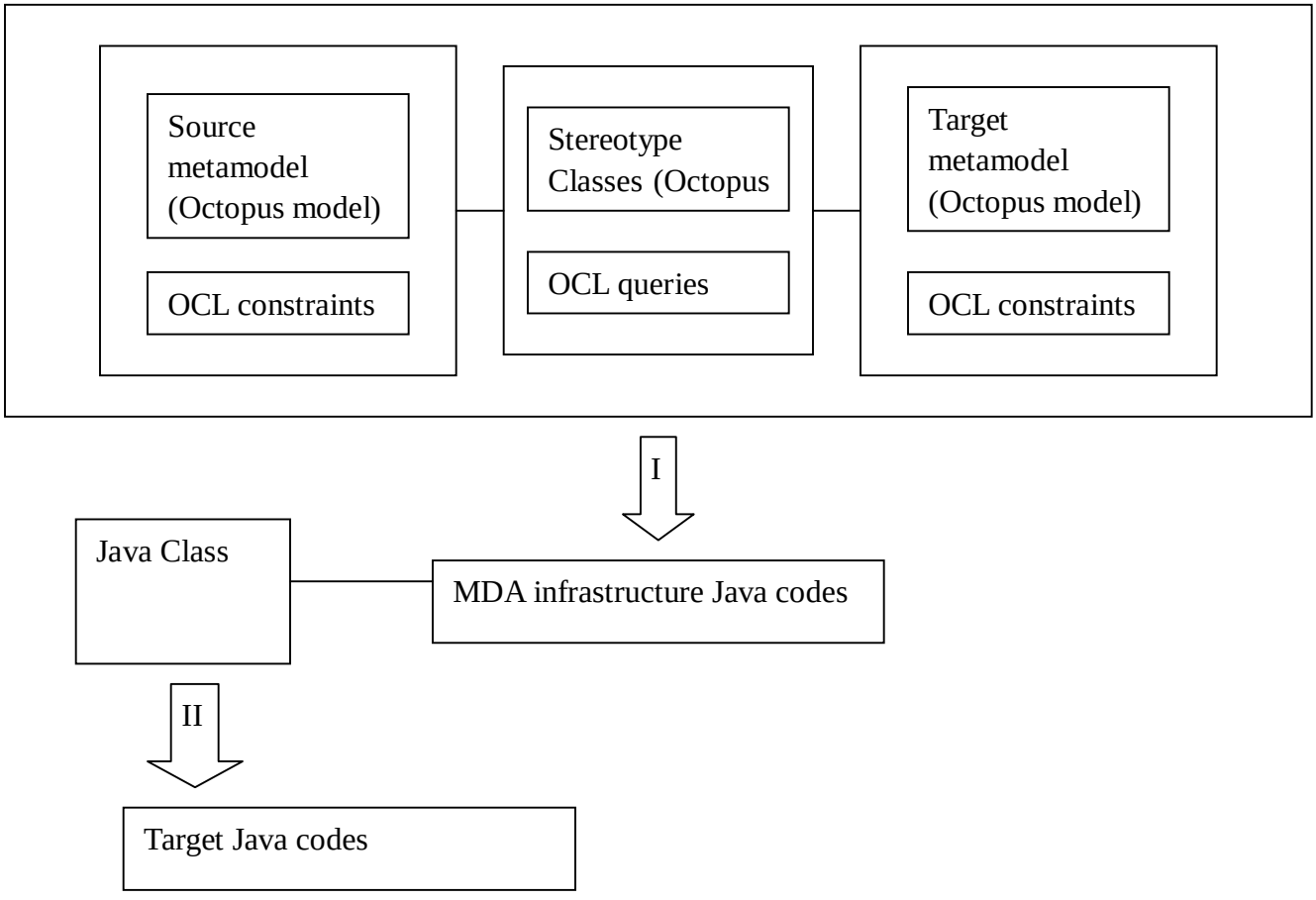
In the same way, to generate JBoss login-config definition, just call

```
wsEndpoint.generateJBossLoginConf();
```

5 Conclusion

Octopus is an OCL expression checking tool. It is able to transform the UML model, including the OCL expressions, into Java code also. In this thesis, Octopus was used as a MDA transformation engine. And OCL query worked as code generator.

As we have known, source metamodel, target metamodel and a transformation engine with transformation rule are building blocks of a MDA transformation. The following diagram shows how Octopus worked in a MDA infrastructure.



As the diagram shows, the steps to build the MDA infrastructure are:

1) *Model the source and target metamodels in Octopus, the Octopus models also include OCL constraints definitions.*

2) *Build a bridge to establish the connection between source and target Octopus models. In fact, this step will build a stereotype. Operations which are responsible for generation final codes should be defined in the stereotype class(es) as method(s). The method(s) should have detailed OCL queries corresponded so that Octopus can generate the code-generator method(s) with implementations.*

3) *Having the whole models in hand, the MDA infrastructure java codes can be generated by Octopus automatically. The arrow with I in the diagram implies the generation procedure.*

4) *To get the final source codes, just adding a Java class with Main method and instantiating the stereotype class and calling the source codes generation methods declared in the stereotype class. This procedure is what arrow II indicates in the diagram above.*

6 Appendix

6.1 Statechart metamodel

6.1.1 Octopus model

```
<package> statechart
<import> statemachine;
+ <class> SMStereotype
<operations>
+getJavaForInterface():String;

<endclass>
<class> State
<attributes>
+ name: String;
+ isInitial: Boolean;
<endclass>

<class> OnEntry<endclass>
<class> Transition<endclass>

<associations>
+ SMStereotype.sm [1] <-> StateMachineInterface.interface[1];
+ SMStereotype.sm [1] <-> State.states[1..*];
+ Transition.outgoing [0..*] <-> + State.start [1];
+ Transition.incoming [0..*] <-> + State.target [1];
+ State.state [1..*] <-> + OnEntry.onEntry [1];

<endpackage>
```

6.1.2 Constraints in OCL

```
package statechart
/*************************************************************
```

```
 Only one state could be initial state
************************************************************/
context State
  inv onlyOneInitial:
    self.allInstances()->one(isInitial )

/************************************************************
 no duplicate names for states
************************************************************/
context State
  inv uniquename :
     self.allInstances()->isUnique(s: State| s.name)

/************************************************************
 no isolated state, one state either has an incoming or outgoing transition
************************************************************/
context State
  inv notisolated:
     (not(self.incoming->isEmpty())) or (not(self.outgoing->isEmpty()))
endpackage – statechart
```

6.2 Java metamodel

```
<package>javamodel
 + <class> CompilationUnit
        <operations>
        + toJavaSource():String;
    <endclass>

 + <class> PackageDeclaration
        <operations>
        + toJavaSource():String;
    <endclass>

 + <class> ImportDeclaration
        <operations>
        + toJavaSource():String;
       <endclass>
```

```
+  <abstract> <class> BodyDeclaration
        <attributes>
        + modifiers:OrderedSet(Modifier);
       <endclass>
+  <class> AnnotationTypeMemberDeclaration <specializes> BodyDeclaration <endclass>

+  <abstract> <class> AbstractTypeDeclaration <specializes> BodyDeclaration
        <operations>
            + toJavaSource(): String;
   <endclass>

+  <class> AnnotationTypeDeclaration  <specializes> AbstractTypeDeclaration
<endclass>

+ <class> TypeDeclaration <specializes> AbstractTypeDeclaration
        <attributes>
        + isInterface :Boolean;
        <operations>
        + toJavaSource(): String;
        + getFields():OrderedSet(FieldDeclaration);
        + getMethods():OrderedSet(MethodDeclaration);
  <endclass>

+ <class> MethodDeclaration <specializes> BodyDeclaration
    <attributes>
    + isConstructor : Boolean;
    <operations>
    + isVarargs():Boolean;
    + parameters():OrderedSet(SingleVariableDeclaration);
    + toJavaSource():String;
  <endclass>
+ <class> FieldDeclaration <specializes> BodyDeclaration
        <operations>
        + fragments():OrderedSet(VariableDeclarationFragment);
  <endclass>

+ <class> Modifier
    <attributes>
    + name:String;
    <endclass>
+ <class> PublicModifier <specializes> Modifier <endclass>
```

```
 + <class> ProtectedModifier  <specializes> Modifier <endclass>
 + <class> PrivateModifier <specializes> Modifier <endclass>
 + <class> StaticModifier <specializes> Modifier <endclass>
 + <class> AbstractModifier <specializes> Modifier <endclass>
 + <class> FinalModifier <specializes> Modifier <endclass>
 + <class> NativeModifier <specializes> Modifier <endclass>
 + <class> SynchronizedModifier <specializes> Modifier <endclass>
 + <class> TransiteModifier <specializes> Modifier <endclass>
 + <class> VolatileModifier <specializes> Modifier <endclass>
 + <class> StrictfpModifier <specializes> Modifier <endclass>

+  <abstract> <class> Name <specializes> Expression
    <attributes>
    -index : Integer;  -- This index represents the position inside a qualified name.
    <operations>
    +isSimpleName():Boolean;
    +isQualifiedName():Boolean;
    +getFullyQualifiedName():String;
    +toJavaSource():String;
    <endclass>
 + <class> QualifiedName <specializes> Name
        <attributes>
        <operations>

   <endclass>

 + <class> SimpleName <specializes> Name
        <attributes>
        + identifier : String;
        <operations>
        + toJavaSource():String;
   <endclass>
 + <class> Annotation <specializes> Expression
    <operations>
    + isModifier():Boolean;
    + isAnnotation():Boolean;
    + isNormalAnnotation():Boolean;
    + isSingleMemberAnnotation():Boolean;
    + toJavaSource():String;
 <endclass>
```

```
+ <class> StringLiteral <specializes> Expression
   <attributes>
       + literalValue:String;
       + escaptedValue:String;
   <operations>
   + toJavaSource() :String;
<endclass>
+ <class> ArrayInitializer <specializes> Expression
   <operations>
   + toJavaSource() :String;
<endclass>
+ <class> NormalAnnotation <specializes> Annotation
   <operations>
       + toJavaSource():String;
<endclass>
+ <class> SingleMemberAnnotation <specializes> Annotation
   <operations>
       + toJavaSource():String;
<endclass>

+ <abstract> <class> Expression
   <operations>
   + toJavaSource():String;
  <endclass>
+ <class> MemberValuePair <endclass>

+ <class> EnumDeclaration <specializes> AbstractTypeDeclaration
   <operations>
   + toJavaSource() :String;
   <endclass>
+ <class> EnumConstantDeclaration <specializes> BodyDeclaration
   <endclass>

+ <class> Statement   <endclass>

+ <class> Block <specializes> Statement
   <operations>
   + toJavaSource():String;
 <endclass>
+ <abstract> <class>  Type
   <operations>
```

```
       + isArrayType():Boolean ;
       + isParameterizedType() :Boolean;
       + isPrimitiveType() :Boolean;
       + isQualifiedType() :Boolean;
       + isSimpleType() :Boolean;
       + isWildcardType()  :Boolean;
       + toJavaSource():String;
 <endclass>

+ <class> PrimitiveType <specializes> Type
   <attributes>
      +name:String;
   <operations>
   + toJavaSource():String;
<endclass>

+ <class>IntType<specializes>PrimitiveType<endclass>
+ <class>ByteType<specializes>PrimitiveType<endclass>
+ <class>ShortType<specializes>PrimitiveType<endclass>
+ <class>CharType<specializes>PrimitiveType<endclass>
+ <class>LongType<specializes>PrimitiveType<endclass>
+ <class>FloatType<specializes>PrimitiveType<endclass>
+ <class>DoubleType<specializes>PrimitiveType<endclass>
+ <class>BooleanType<specializes>PrimitiveType<endclass>
+ <class>VoidType<specializes>PrimitiveType <endclass>

+ <class> ArrayType <specializes> Type
   <operations>
   + toJavaSource():String;
<endclass>
+ <class> QualifiedType <specializes> Type
   <operations>
   + toJavaSource():String;
<endclass>
+ <class> SimpleType <specializes> Type
   <operations>
   + toJavaSource():String;
<endclass>
+ <class> ParameterizedType <specializes> Type
   <operations>
   + toJavaSource():String;
```

```
 <endclass>
 + <class> WildcardType <specializes> Type
    <operations>
    + toJavaSource():String;
 <endclass>

 + <class> VariableDeclaration
    <attributes>
    - extraArrayDimensions: Integer;
 <endclass>

 + <class> VariableDeclarationFragment <specializes> VariableDeclaration
        <attributes>
        - extraArrayDimensions: Integer;
<endclass>

 + <class> SingleVariableDeclaration <specializes> VariableDeclaration
    <attributes>
     + variableArity :Boolean;
     - modifiers:OrderedSet(Modifier);
     - extraArrayDimensions:Integer;
 <endclass>

<associations>
 + QualifiedName.qualifiedname [1]  ->  + SimpleName.namepart [1] ;
 + QualifiedName.qualifiedname [1]  ->  + Name.qualifier [0..1] ;

 + PackageDeclaration.packageDeclaration [1] <aggregate> ->  + Name.packageName
[1]   ;
 + PackageDeclaration.packageDeclaration [1] <aggregate> ->  +
Annotation.annotations[1..*]<ordered>   ;

 + Annotation.annotation[1] <aggregate> -> Name.typeName[1];
 + NormalAnnotation.normalAnnotation [1] <aggregate> ->
MemberValuePair.values[0..*]<ordered>;
 + MemberValuePair.pair[1] -> SimpleName.name[1];
 + MemberValuePair.pair[1] -> Expression.value[1];
 + SingleMemberAnnotation.singleAnn[1] -> Expression.value[1];

 + BodyDeclaration.bodyDeclaration [1]   -> Annotation.annotation [0..1] ;
+ ImportDeclaration.import [1] <aggregate>  -> Name.name [1] ;
```

```
 + CompilationUnit.unit [1]  ->  + PackageDeclaration.packageInfo [0..1]   ;
 + CompilationUnit.unit [1] <aggregate>  -> ImportDeclaration.imports [0..*]
<ordered>;
 + CompilationUnit.unit [1] <composite>  -> AbstractTypeDeclaration.types [1..*]
<ordered>;

 + TypeDeclaration.type [1] <aggregate>  -> BodyDeclaration.bodyDeclarations [0..*]
<ordered>; --BodyDeclaration is a supertype, it could be fielddeclaration or
methoddeclaration--
 + AnnotationTypeMemberDeclaration.annotationTypeMember[1]
    <aggregate> -> Name.memberName[1];
  + AnnotationTypeMemberDeclaration.annotationTypeMember[1]
    <aggregate> -> Type.memberType[1];
  + AnnotationTypeMemberDeclaration.annotationTypeMember[1]
    <aggregate> -> Expression.optionalDefaultValue[0..1];

  + ArrayInitializer.array[1] <aggregate> -> Expression.expressions[0..*]<ordered>;

 + TypeDeclaration.type [1]  <aggregate> -> Type.superInterfaceTypes[0..*]<ordered>;
 + TypeDeclaration.type [1]  <aggregate> -> Type.superClassType[0..1];

 + EnumDeclaration.enumDeclaration[1] <composite> ->
EnumConstantDeclaration.enumConstants[0..*]<ordered>;
 + EnumConstantDeclaration.enumDeclaration[1]  -> SimpleName.constantName[1];

 + AbstractTypeDeclaration.type [1]  <aggregate> -> SimpleName.typeName [1] ;
 + AbstractTypeDeclaration.type [1]  <aggregate> -> BodyDeclaration.bodyDeclarations
[0..*]<ordered> ;

 + VariableDeclaration.<noName> [1]  -> SimpleName.name [1] ;
 + VariableDeclarationFragment.fragment [1] <aggregate> -> Name.variableName[1];

 + MethodDeclaration.methodDeclaration [1]  -> SimpleName.name [1] ;
 + MethodDeclaration.methodDeclaration [1]  -> Type.returnType [1] ;
 + MethodDeclaration.methodDeclaration [1] <aggregate>  ->
SingleVariableDeclaration.parameters[0..*] <ordered>;
 + MethodDeclaration.methodDeclaration [1]  -> Block.methodBody [0..1] ;
 + MethodDeclaration.methodDeclaration [1] -> Name.thrownExceptions[0..*]<ordered>;

 + SingleVariableDeclaration.variable [1] <aggregate> -> SimpleName.variableName[1];
```

```
 + SingleVariableDeclaration.variable [1] <aggregate> -> Type.type[1];
 + SingleVariableDeclaration.variable[1] -> Expression.Initializer[0..1];

 + Block.block[1] -> Statement.statements[0..*]<ordered>;

 + FieldDeclaration.field [1]  <aggregate> -> SimpleName.name [1] ;
 + FieldDeclaration.field [1]  <aggregate> -> Type.baseType [1] ;
 + FieldDeclaration.field [1]  <aggregate> ->
VariableDeclarationFragment.variableDeclarationFragments [1..*]<ordered>;

 + SimpleType.simplyType [1]  -> SimpleName.name[1];
 + QualifiedType.qulifiedType [1]  -> SimpleName.name[1];
 + QualifiedType.qulifiedType [1]  -> Type.qualifier[1];

 + PrimitiveType.primitiveType[1] <composite> -> IntType.intType[1];
 + PrimitiveType.primitiveType[1] <composite> -> ByteType.byteType[1];
 + PrimitiveType.primitiveType[1] <composite> -> ShortType.shortType[1];
 + PrimitiveType.primitiveType[1] <composite> -> CharType.charType[1];
 + PrimitiveType.primitiveType[1] <composite> -> LongType.longType[1];
 + PrimitiveType.primitiveType[1] <composite> -> FloatType.floatType[1];
 + PrimitiveType.primitiveType[1] <composite> -> BooleanType.booleanType[1];
 + PrimitiveType.primitiveType[1] <composite> -> VoidType.voidType[1];

 + Modifier.modifier[1] <composite> -> PublicModifier.publicModifier[1];
 + Modifier.modifier[1] <composite> -> ProtectedModifier.protectedModifier[1];
 + Modifier.modifier[1] <composite> -> PrivateModifier.privateModifier[1];
 + Modifier.modifier[1] <composite> -> StaticModifier.staticModifier[1];
 + Modifier.modifier[1] <composite> -> AbstractModifier.abstractModifier[1];
 + Modifier.modifier[1] <composite> -> FinalModifier.finalModifier[1];
 + Modifier.modifier[1] <composite> -> NativeModifier.nativeModifier[1];
 + Modifier.modifier[1] <composite> -> SynchronizedModifier.synchronizedModifier[1];
 + Modifier.modifier[1] <composite> -> TransiteModifier.transiteModifier[1];
 + Modifier.modifier[1] <composite> -> VolatileModifier.volatileModifier[1];
 + Modifier.modifier[1] <composite> -> StrictfpModifier.strictfpModifier[1];


<endpackage>
```

6.3 Main class for generating Microwave oven Java codes

```
import javamodel.internal.*;
import statechart.internal.*;
import statemachine.internal.*;
import java.util.*;
import java.io.*;

public class Test {

    // modifiers
    private final Modifier PUBLIC = new PublicModifier();
    private final Modifier PROTECTED = new ProtectedModifier();
    private final Modifier PRIVATE = new PrivateModifier();
    private final Modifier STATIC = new StaticModifier();
    private final Modifier ABSTRACT = new AbstractModifier();
    private final Modifier FINAL = new FinalModifier();
    private final Modifier NATIVE = new NativeModifier();
    private final Modifier SYNCHRONIZED = new SynchronizedModifier();
    private final Modifier TRANSITE = new TransiteModifier();
    private final Modifier VOLATILE = new VolatileModifier();
    private final Modifier STRICTFP = new StrictfpModifier();

    // primitive types
    private final Type INT = new IntType();
    private final Type BYTE = new ByteType();
    private final Type SHORT = new ShortType();
    private final Type CHAR = new CharType();
    private final Type LONG = new LongType();
    private final Type FLOAT = new FloatType();
    private final Type BOOLEAN = new BooleanType();
    private final Type VOID = new VoidType();
    public static void main(String[] args) {
        Test test = new Test();
        test.doMDA();
    }

    public void doMDA() {
```

```
        // #create methods public void openDoor()
        // ... method without "i" is for the class using, will be assigned later
        MethodDeclaration openDoor = new MethodDeclaration();
        Type returnType = new PrimitiveType();
        ArrayList modifiers = new ArrayList();
        modifiers.add(this.PUBLIC);
        MethodDeclaration iopenDoor = getMethodDeclaration(modifiers,
                "openDoor", this.VOID);

        // #create methods public void closeDoor()
        // ... method without "i" is for the class using, will be assigned later
        MethodDeclaration closeDoor = new MethodDeclaration();
        modifiers = new ArrayList();
        modifiers.add(this.PUBLIC);
        MethodDeclaration icloseDoor = getMethodDeclaration(modifiers,
                "closeDoor", this.VOID);

        // $$$ create states and add them into sms
        State s_DOOROPEN = new State();
        s_DOOROPEN.setName("DOOROPEN");
        State s_READYTOCOOK = new State();
        s_READYTOCOOK.setName("READYTOCOOK");
        s_READYTOCOOK.setInitial(true);
        State s_COOKING = new State();
        s_COOKING.setName("COOKING");
        State s_COOKINGINTERRUPTED = new State();
        s_COOKINGINTERRUPTED.setName("COOKINGINTERRUPTED");
        State s_COOKINGCOMPLETE = new State();
        s_COOKINGCOMPLETE.setName("COOKINGCOMPLETE");
        State s_COOKINGEXTENDED = new State();
        s_COOKINGEXTENDED.setName("COOKINGEXTENDED");

        // $$$ create the enum type with states
        EnumDeclaration states = this.getStatesEnum("States");
        states.addToEnumConstants(this.getEnumItem(s_DOOROPEN));
        states.addToEnumConstants(this.getEnumItem(s_READYTOCOOK));
        states.addToEnumConstants(this.getEnumItem(s_COOKING));
        states.addToEnumConstants(this.getEnumItem(s_COOKINGINTERRUPTED));
        states.addToEnumConstants(this.getEnumItem(s_COOKINGCOMPLETE));
        states.addToEnumConstants(this.getEnumItem(s_COOKINGEXTENDED));
```

```
        // $$$ create the statemachineinterface - IMicrowave
        StateMachineInterface iMicrowave = getStateMachineInterface("IMicrowave");

        // $$$ create statespaces annotation, and adding to IMicrowave
        NormalAnnotation stateSpaces = getStateSpaceAnnotation(states,
                s_READYTOCOOK);
        iMicrowave.setAnnotation(stateSpaces);
        // $$$ adding methods into interface
        iMicrowave.addToBodyDeclarations(iopenDoor);
        iMicrowave.addToBodyDeclarations(icloseDoor);

        // $$$ create Microwave Class - main business class
        TypeDeclaration microwaveCLS = getMainClass("Microwave", iMicrowave);

        // ...adding constructor
        MethodDeclaration constructor = getConstructor(microwaveCLS);
        microwaveCLS.addToBodyDeclarations(constructor);

        // $$$ createing all transition objects
        Transition tr_READYTOCOOK_DOOROPEN = getTransition(s_READYTOCOOK,
                s_DOOROPEN);
        Transition tr_COOKING_COOKINGINTERRUPTED = getTransition(s_COOKING,
                s_COOKINGINTERRUPTED);
        Transition tr_COOKINGCOMPLETE_DOOROPEN = getTransition(
                s_COOKINGCOMPLETE, s_DOOROPEN);
        Transition tr_COOKINGEXTENDED_COOKINGINTERRUPTED = getTransition(
                s_COOKINGEXTENDED, s_COOKINGINTERRUPTED);
        Transition tr_DOOROPEN_READYTOCOOK = getTransition(s_DOOROPEN,
                s_READYTOCOOK);
        Transition tr_COOKINGINTERRUPTED_READYTOCOOK = getTransition(
                s_COOKINGINTERRUPTED, s_READYTOCOOK);

        // $$$ creating all transition annotations from transition
        Annotation trAnn_READYTOCOOK_DOOROPEN = getTransitionAnnotation(states,
                tr_READYTOCOOK_DOOROPEN);
        Annotation trAnn_COOKING_COOKINGINTERRUPTED = getTransitionAnnotation(
                states, tr_COOKING_COOKINGINTERRUPTED);
        Annotation trAnn_COOKINGCOMPLETE_DOOROPEN = getTransitionAnnotation(
                states, tr_COOKINGCOMPLETE_DOOROPEN);
        Annotation trAnn_COOKINGEXTENDED_COOKINGINTERRUPTED =
getTransitionAnnotation(
```

```
            states, tr_COOKINGEXTENDED_COOKINGINTERRUPTED);
        Annotation trAnn_DOOROPEN_READYTOCOOK = getTransitionAnnotation(states,
            tr_DOOROPEN_READYTOCOOK);
        Annotation trAnn_COOKINGINTERRUPTED_READYTOCOOK = getTransitionAnnotation(
            states, tr_COOKINGINTERRUPTED_READYTOCOOK);

        // ... adding openDoor method together with the annotation
        Annotation annOpenDoor = getMethodAnnotation(new Annotation[] {
            trAnn_READYTOCOOK_DOOROPEN, trAnn_COOKING_COOKINGINTERRUPTED,
            trAnn_COOKINGCOMPLETE_DOOROPEN,
            trAnn_COOKINGEXTENDED_COOKINGINTERRUPTED });

        // ... adding closeDoor method together with the annotation
        Annotation annCloseDoor = getMethodAnnotation(new Annotation[] {
            trAnn_DOOROPEN_READYTOCOOK,
            trAnn_COOKINGINTERRUPTED_READYTOCOOK });

        closeDoor.setAnnotation(annCloseDoor);
        openDoor.setAnnotation(annOpenDoor);

        // $$$ create SMStereotype ins -> sms
        SMStereotype sms = new SMStereotype();
        // $$$ ADD STATES and interface INTO SMS
        sms.addToStates(s_DOOROPEN);
        sms.addToStates(s_READYTOCOOK);
        sms.addToStates(s_COOKING);
        sms.addToStates(s_COOKINGINTERRUPTED);
        sms.addToStates(s_COOKINGCOMPLETE);
        sms.addToStates(s_COOKINGEXTENDED);

        // assign the methods for class using, actually just add a empty method
        // body
        // then add those assigned methods into class
        openDoor = this.getEmptyMethodImplementation(openDoor, iopenDoor);
        closeDoor = this.getEmptyMethodImplementation(closeDoor, icloseDoor);

        microwaveCLS.addToBodyDeclarations(openDoor);
        microwaveCLS.addToBodyDeclarations(closeDoor);

        CompilationUnit statesUnit = new CompilationUnit();
        CompilationUnit iMicrowaveUnit = new CompilationUnit();
```

```
        CompilationUnit microwaveCLSUnit = new CompilationUnit();

        statesUnit.setPackageInfo(getPackage());
        statesUnit.addToTypes(states);

        iMicrowaveUnit.setPackageInfo(getPackage());
        iMicrowaveUnit.addToTypes(iMicrowave);

        microwaveCLSUnit.setPackageInfo(getPackage());
        microwaveCLSUnit.addToTypes(microwaveCLS);

        try {

            this.write("src\\generated\\States.java", statesUnit
                            .toJavaSource());
            this.write("src\\generated\\IMicrowave.java", iMicrowaveUnit
                    .toJavaSource());
            this.write("src\\generated\\Microwave.java", microwaveCLSUnit
                    .toJavaSource());

            System.out
                    .println("pls check the src/generated package for the generated
java source");
        } catch (IOException ex) {
            ex.printStackTrace();
        }

    }

    private void write(String fileName, String text) throws IOException {
        PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(
                fileName)));

        out.print(text);
        out.close();
    }

    private PackageDeclaration getPackage() {
        PackageDeclaration myPackage = new PackageDeclaration();
        SimpleName packageName = new SimpleName();
        packageName.setIdentifier("generated");
```

```
        myPackage.setPackageName(packageName);
        return myPackage;
    }

    private MethodDeclaration getEmptyMethodImplementation(
            MethodDeclaration resultMethod, MethodDeclaration interfaceMethod) {

        interfaceMethod.copyInfoInto(resultMethod);
        Block blk = new Block();
        resultMethod.setMethodBody(blk);
        return resultMethod;
    }

    private Transition getTransition(State startState, State targetState) {
        Transition tr = new Transition();

        tr.setStart(startState);
        tr.setTarget(targetState);
        return tr;

    }

    private StateMachineInterface getStateMachineInterface(String identifier) {
        StateMachineInterface stateInterface = new StateMachineInterface();

        // interface modifer
        ArrayList interfaceModifiers = new ArrayList();
        interfaceModifiers.add(this.PUBLIC);
        // name
        SimpleName interfaceName = new SimpleName();
        interfaceName.setIdentifier(identifier);

        stateInterface.setInterface(true);
        stateInterface.setTypeName(interfaceName);
        stateInterface.setModifiers(interfaceModifiers);

        return stateInterface;

    }

    private MethodDeclaration getMethodDeclaration(List modifiers,
```

```
            String identfier, Type returnType) {
        MethodDeclaration method = new MethodDeclaration();

        SimpleName sn = new SimpleName();
        sn.setIdentifier(identfier);

        method.setConstructor(false);
        method.setModifiers(modifiers);
        method.setName(sn);
        method.setReturnType(returnType);

        return method;
    }

    private EnumDeclaration getStatesEnum(String typeName) {
        EnumDeclaration states = new EnumDeclaration();
        ArrayList modifiers = new ArrayList();
        modifiers.add(this.PUBLIC);
        states.setModifiers(modifiers);
        SimpleName enumName = new SimpleName();
        enumName.setIdentifier(typeName);
        states.setTypeName(enumName);
        return states;
    }

    private EnumConstantDeclaration getEnumItem(State state) {
        EnumConstantDeclaration item = new EnumConstantDeclaration();
        SimpleName itemName = new SimpleName();
        itemName.setIdentifier(state.getName());
        item.setConstantName(itemName);
        return item;
    }

    private TypeDeclaration getMainClass(String identifier,
            TypeDeclaration implementedInterface) {
        TypeDeclaration mainClass = new TypeDeclaration();
        ArrayList classModifiers = new ArrayList();
        classModifiers.add(this.PUBLIC);
        // name
        SimpleName className = new SimpleName();
        className.setIdentifier(identifier);
```

```
    mainClass.setInterface(false);
    mainClass.setTypeName(className);
    mainClass.setModifiers(classModifiers);

    // set implemented interface(s)
    SimpleName interfaceName = (SimpleName) implementedInterface
            .getTypeName();
    ArrayList superInterfaces = new ArrayList();
    SimpleType superInterface = new SimpleType();
    superInterface.setName(interfaceName);
    superInterfaces.add(superInterface);
    mainClass.setSuperInterfaceTypes(superInterfaces);
    return mainClass;
}

private MethodDeclaration getConstructor(TypeDeclaration cls) {
    MethodDeclaration constructor = new MethodDeclaration();
    constructor.setConstructor(true);
    constructor.addToModifiers(this.PUBLIC);
    constructor.setName(cls.getTypeName());
    SimpleType returnType = new SimpleType();
    SimpleName sn = new SimpleName();
    sn.setIdentifier("");
    returnType.setName(sn);
    Block emptyBlock = new Block();
    constructor.setMethodBody(emptyBlock);
    constructor.setReturnType(returnType);
    return constructor;
}

/*
 * @Statespace ( states = { "READYTOCOOK", "DOOROPEN", "COOKING",
 * "COOKINGINTERRUPTED", "COOKINGCOMPLETE", "COOKINGEXTENDED" }, initial =
 * "READYTOCOOK" )
 */
private NormalAnnotation getStateSpaceAnnotation(EnumDeclaration stateEnum,
        State initState) {
    NormalAnnotation ss = new NormalAnnotation();
    SimpleName ssTypeName = new SimpleName();
    ssTypeName.setIdentifier("StateSpace");
```

```
        ss.setTypeName(ssTypeName);
        MemberValuePair mvp1 = new MemberValuePair();
        MemberValuePair mvp2 = new MemberValuePair();
        SimpleName memberName1 = new SimpleName();
        SimpleName memberName2 = new SimpleName();
        memberName1.setIdentifier("states");
        memberName2.setIdentifier("initial");
        mvp1.setName(memberName1);
        mvp2.setName(memberName2);
        StringLiteral string1 = new StringLiteral();
        StringLiteral string2 = new StringLiteral();
        String spaces = "";
        String init = "\"" + initState.getName() + "\"";
        List stateEnumList = stateEnum.getEnumConstants();
        Iterator it = stateEnumList.iterator();
        EnumConstantDeclaration tmp = null;
        while (it.hasNext()) {
            tmp = (EnumConstantDeclaration) it.next();
            if (spaces.equals(""))
                spaces += "{\"" + tmp.getConstantName().getIdentifier() + "\"";
            else
                spaces += ",\"" + tmp.getConstantName().getIdentifier() + "\"";

        }
        spaces += "}";
        string1.setLiteralValue(spaces);
        string2.setLiteralValue(init);
        mvp1.setValue(string1);
        mvp2.setValue(string2);
        ss.addToValues(mvp1);
        ss.addToValues(mvp2);
        return ss;
    }

    private SingleMemberAnnotation getMethodAnnotation(Annotation[] trans) {
        SingleMemberAnnotation annM = new SingleMemberAnnotation();
        SimpleName tranTypeName = new SimpleName();
        tranTypeName.setIdentifier("OutgoingTransitions");
        annM.setTypeName(tranTypeName);
        ArrayInitializer array = new ArrayInitializer();
        ArrayList tranList = new ArrayList();
```

```
        for (int i = 0; i < trans.length; i++) {
            tranList.add(trans[i]);
        }
        array.setExpressions(tranList);
        annM.setValue(array);
        return annM;
    }

    private NormalAnnotation getTransitionAnnotation(
            EnumDeclaration statesEnum, Transition tran) {
        State stateFrom = (State) tran.getStart();
        State stateTo = (State) tran.getTarget();
        NormalAnnotation tran1 = new NormalAnnotation();
        SimpleName tranTypeName = new SimpleName();
        tranTypeName.setIdentifier("Transition");
        tran1.setTypeName(tranTypeName);
        MemberValuePair mvp1 = new MemberValuePair();
        MemberValuePair mvp2 = new MemberValuePair();
        SimpleName memberName1 = new SimpleName();
        SimpleName memberName2 = new SimpleName();
        memberName1.setIdentifier("from");
        memberName2.setIdentifier("to");
        mvp1.setName(memberName1);
        mvp2.setName(memberName2);
        StringLiteral string1 = new StringLiteral();
        StringLiteral string2 = new StringLiteral();
        string1.setLiteralValue(statesEnum.getTypeName().getIdentifier() + "."
                + stateFrom.getName());
        string2.setLiteralValue(statesEnum.getTypeName().getIdentifier() + "."
                + stateTo.getName());
        mvp1.setValue(string1);
        mvp2.setValue(string2);
        tran1.addToValues(mvp1);
        tran1.addToValues(mvp2);
        return tran1;
    }
}
```

7 Bibliography

[KWB03] Anneke Kleppe, Jos Warmer & Wim Bast "*MDA Explained. The Model Driven Architecture: Practice and Promise"* published by Addison Wesley 2003

[WK03] Jos Warmer, Anneke Kleppe *"Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition"* published by Addison Wesley 2003

[Kla05] Klasse Objecten company *internet at http://www.klasse.nl/english,* 2005

[Mod] ModFact "*QVT Project*" *internet at http://modfact.lip6.fr/qvtP.html*

[LBD] Torsten Lodderstedt, David Basin, and Jürgen Doser "*SecureUML: A UML-Based Modeling Language for Model-Driven Security* "

[IO] Interactive Object "*ArcStyler Cartridge Guide for MDA-Security* "

[JBo05] JBoss Inc. *"WSSecureEndpoint" internet at http://wiki.jboss.org/wiki/Wiki.jsp?page=WSSecureEndpoint,2005*