International Journal of Software Engineering and Its Applications
Vol.8, No.7 (2014), pp.73-94
http://dx.doi.org/10.14257/ijseia.2014.8.7.07

Model-Driven Generation: From Models to MVC2 Web Applications

M'hamed Rahmouni[1] and Samir Mbarki[2]

[1,2]Department of Computer Science, Faculty of Science, Ibn Tofail University, Kenitra, BP 133, Morocco
[1]md.rahmouni@yahoo.fr, [2]mbarkisamir@hotmail.com

Abstract

Transformation is one of the prominent features and the rising research area of MDA (Model Driven Architecture) since last few years. MDA is a new paradigm of software engineering that considers models as first-class entities. Indeed, techniques of model transformations may be implemented to automatically generate elements of the system from models. In this paper, we have conducted a new technique of model transformation to generate the code of web application. This technique is based on the combination of diagrams. In this latter, we focus on the combination of the UML class diagram and the UML sequence diagram. This technique of transformation is defined by an algorithm. In this algorithm, we consider only the operations belonging to the two diagrams already combined. Practically, we transform only the operations that have a sequence diagram and belong to the class diagram. This algorithm is appeared in transformation rules. These rules are written in ATL transformation language. The objective of these rules is to generate an MVC2 web model. The generated model is used as an input file of JET template in order to generate anapplication web code. Also; it presents a case study to illustrate this proposal.

Keywords: *MDA, ATL transformation, Code generation, MVC2 web, CIM, PIM, PSM, Metamodel*

1. Introduction

Software development techniques are continuously evolving with the goal of solving the main problems that still affect the building and maintenance of software systems: time, costs and error-proneness [11]. Model-driven architecture (MDA) [1] aims to reduce at least some of these problems. They focus on the construction of models, specification of transformation rules, tool support and automatic generation of code and documentation. The central idea of MDA is to separate the platform independent design from the platform specific implementation of applications delaying as much as possible the dependence on specific technologies [23].

The MDA uses models as first class entities, enabling the definition and automatic execution of transformations between models and from models to code. The creation of metamodels for specifying modeling languages is a basic task in MDA. Models in MDA are the key artefacts in all phases of development and are mostly expressed with Unified Modeling Language (UML) [2, 3]. Also the specification of transformations between models, are called model-to-model (M2M) transformations, and from model to code, are called model-to-text (M2T) transformations. The main advantage of this approach of software development is that MDA tools enable these transformations to be specified and executed automatically, using supporting languages and tools for MDA. This development approach is

ISSN: 1738-9984 IJSEIA

currently being applied to many domains in software development, such as embedded systems, Web engineering, Ontology engineering and more.

The work described here presents a method based on MDA approach to generate MVC2 web model and thereafter generates the applicationweb code from this model. This method is based on the combination of diagrams. The basic idea of this method is to combine the UML class and the sequence diagrams to constitute one source meta-model of the transformation language.

The aim of using the sequence diagram is to know the relation between the Action classes, the Action Form classes and the jsp pages. Through this sequence diagram, we can especially know the input jsp page of another page which it will require. Another important objective of using this diagram is to define the attributes of the Action Form and the object or class to which it belong each attribute.

To know the relation between the classes, methods and attributes of each class as well as associations between classes, we use the UML class diagram. The objective of the combination of this latter and the UML sequence diagram is to generate precisely the elements that constitute the MVC2 Webingredients and the relations between these elements.

To achieve this transformation, we used the ATL (Atlas Transformation Language) transformation language [4-6]. The transformation rules are based on the following idea: each operation generates an Action classes, an Action Form classes and a jsp pages. In the transformationalgorithm, we consider only the CRUD operations and the operations belonging to the combination of CD and SD. After generating the MVC 2 web model, we conduct a process of code generation from this model. This process is assured by using the JET2 [7] template.

The remaining part of this paper is as follows: Section 2 explains the transformation algorithm. Section 3 presents the PIM and PSM metamodels. Section 4 is dedicated to the case study of e-commerce Web applications. Section 5 describes the transformation rules. Section 6 is dedicated to the code generation process. Section 7 presents the result of code generation process. Section 8 evaluates this code generation method. Section 9 is dedicated to the related work. Finally, section 10 concludes the work and gives hints about future work.

2. Transformation Algorithm

In this work, we combine UML class diagram and UML sequence diagram for constitute one source metamodel. The main idea of the transformation algorithm is based on the fact that each operation must belong to these two diagrams. The complete algorithm of transformation rules is as follow:

- Consider the UML class diagram and the UML sequence diagram as an input model.
- Browse the UML class diagram and select an operation.
- Test if this operation belongs to the sequence diagram. If true then transform this operation into PSM code. Else go to another operation without generating the PSM code.
- Afterverifying thatthe operation belongsto the CDandSDsimultaneously, so we transform this operation into an Action class, an Action Form class and a jsp page.

This algorithm is represented by the following Figure.

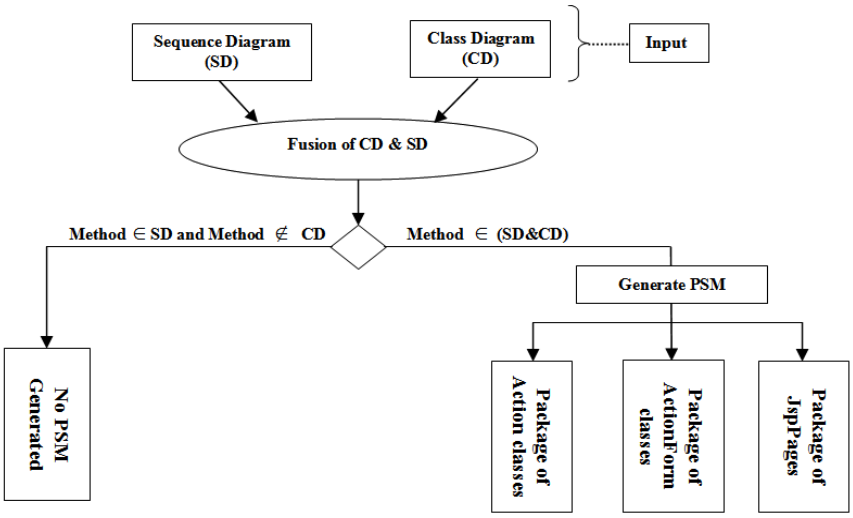

Figure 1. Algorithm for Transforming the Combination of SD and CD into MVC2 Web Model

To implement this algorithm, we begin by the implementation of source and target meta-models.

3. PIM and PSM MetaModels

In this section, we present the different meta-classes that constitute the PIM and PSM metamodels. The PIM is constituted by the combination of UML class diagram (CD) and UML sequence diagram (SD). The PIM and PSM metamodels are detailed here in Figures 2-4.

3.1. Class Diagram Metamodel:

In this metamodel, UML Package corresponds to the concept of UML package, this meta-class is related to the Classifier meta-class. This represents both the concept of UML class and the concept of data type. The Property meta-class expresses the concept of properties of an UML class or references to other classes (uni and bidirectional associations). Figure 2 shows the CD metamodel.

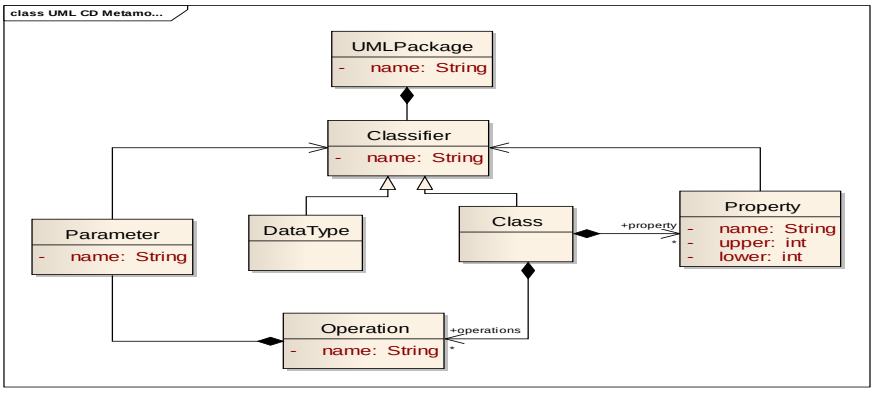

Figure 2. UML Class Diagram Metamodel

3.2. Sequence Diagram Metamodel:

Figure 3 shows our simplified metamodel for UML 2 sequence diagram. A sequence diagram is represented by a set of lifelines. A lifeline has a top-down ordered sequence of occurrences.

An occurrence can be one of five kinds (event, combined Fragment, start, end, arbEvt), where only events or combined fragments conceptually occur on an ordinary sequence diagram lifeline. The meta occurrence of kind start shall be the very first occurrence on a lifeline, and the meta occurrence of kind end shall be the very last occurrence on a lifeline. These meta occurrences enables us to easily specify the replacement of a subsequence of occurrences on a lifeline.

Finally, an occurrence of kind arbEvt represents the lifeline symbol called arbitrary events, which was previously introduced in [19]. This symbol allows matches to have an arbitrary number of occurrences in the symbol's position. Generally, the symbol can be placed anywhere on a lifeline. In this paper we restrict the usage to at most one symbol per lifeline and if used it shall be placed as the very first occurrence on the lifeline. This restriction is sufficient for our transformation from sequence diagrams to state machines, and allows us to focus on the contributions of this paper.

A message consists of a send event and a receive event, which are normally placed on two different lifelines. A combined fragment spans over many lifelines and it has one or more operands. A combined fragment with operator opt, loop or neg contains exactly one operand, while for other operators (*e.g.*, alt, par) it contains an arbitrary number of operands.

Each operand has a guard attribute and spans over a subset of the lifelines which it's combined fragment spans over. An operand lifeline has a part of relation to indicate to which lifeline it belongs.

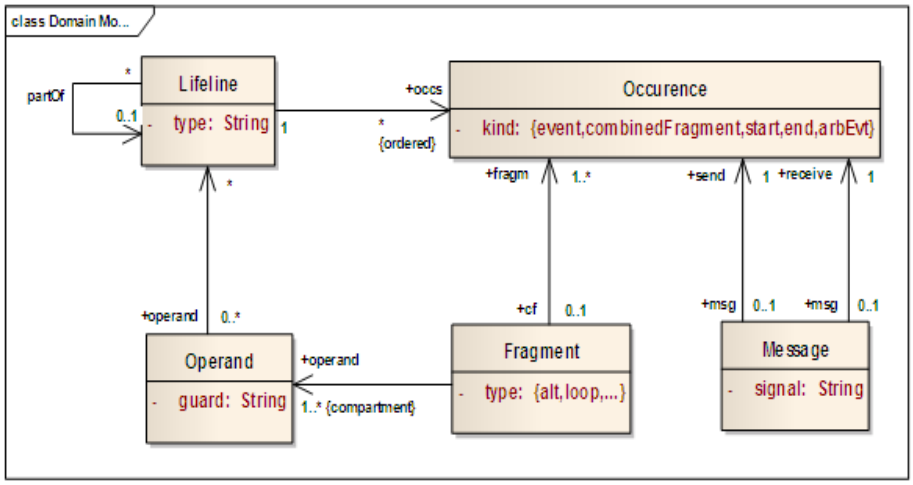

Figure 3. Sequence Diagram metamodel

3.3. Struts Metamodel:

Figure 4 corresponds to the PSM metamodel that is a Struts metamodel target. In this meta-model, we show more interest in the tier controller. The Action Mapping meta-class contains the information deployment for a particular Action class. The Action Form meta-class is a Bean encapsulating the parameters of a form from the view part. The execute () method of Action class performs its processing and then calls the find forward () method on

the mapping object. The return value is an object of an Action Forward type. The Action meta-class represents the concept of secondary controller. The Action classes contain the specific processing of the application. Consequently, they must be related to business classes. The complete target metamodel is detailed in [8].

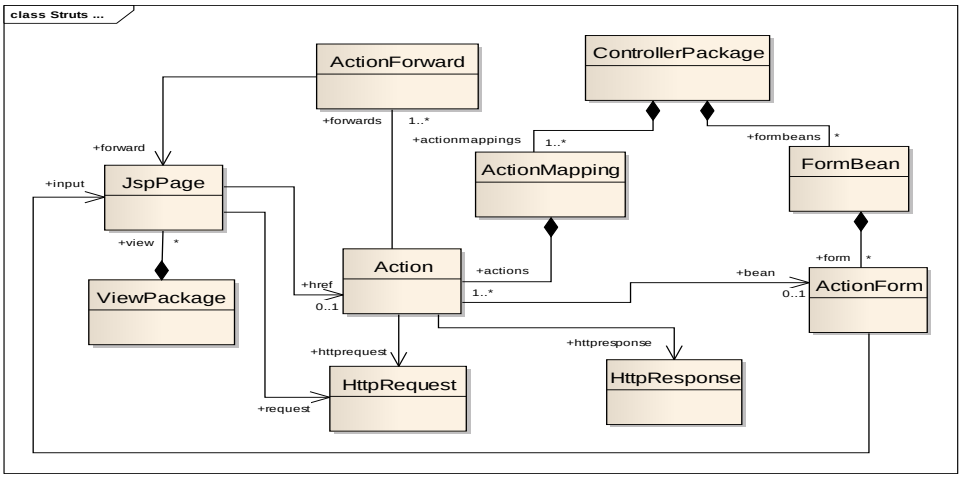

Figure 4. Platform-Specific Struts Model

4. Case Study

In this section, we describe our UML use case and thereafter the UML class diagram. We focus on the source model of this web application. This model is a XMI file which contains all components of this application.

In this case study, each customer could have some web user identity. Web user could be in several states and could be linked to one shopping cart. Each customer has exactly one account. Account owns shopping cart and orders. Orders are sorted and unique. Each order is linked to none to several payments. Each order has current order status. Both order and shopping cart have line items linked to specific product.

Based on the above analysis, we design the use case diagram of the PC online shopping system. In this case we detail only the View Items use case. The Figure 5 shows the View Items use case. The UML source model is detailed in Figure 6.

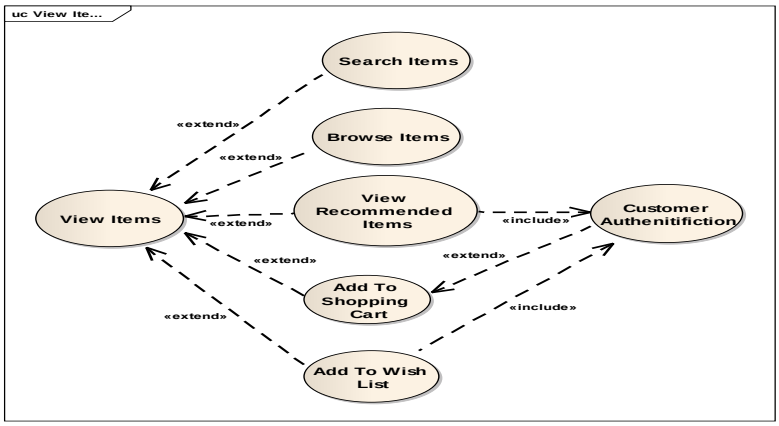

Figure 5. The View Items Use Case

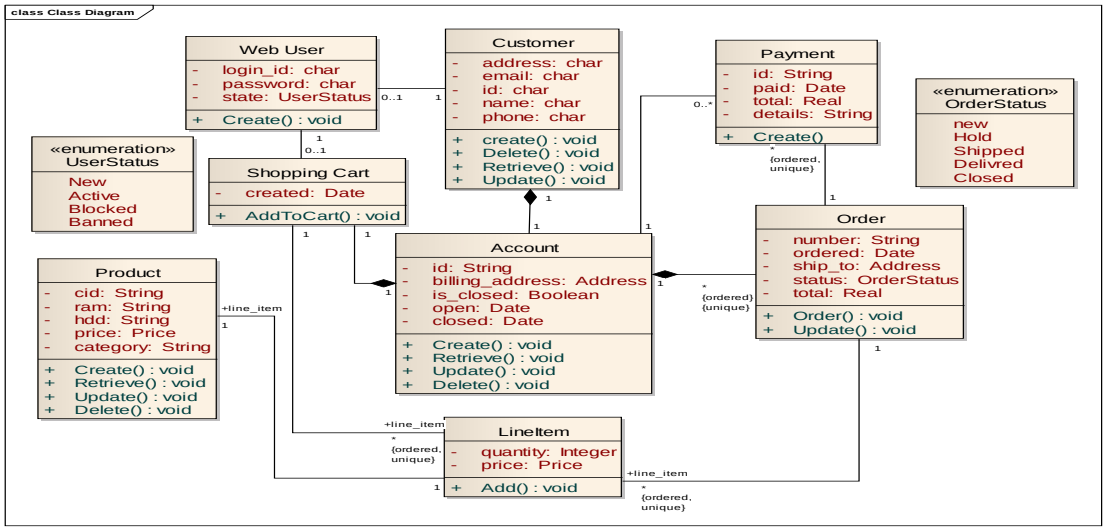

Figure 6. The UML Source Model of the PC Online Shopping

In the following section we show the sequence diagram of the Add to cartuse case. The Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines (Objects) [9]. Figure 7 shows the sequence diagram of the Add to cart use case.

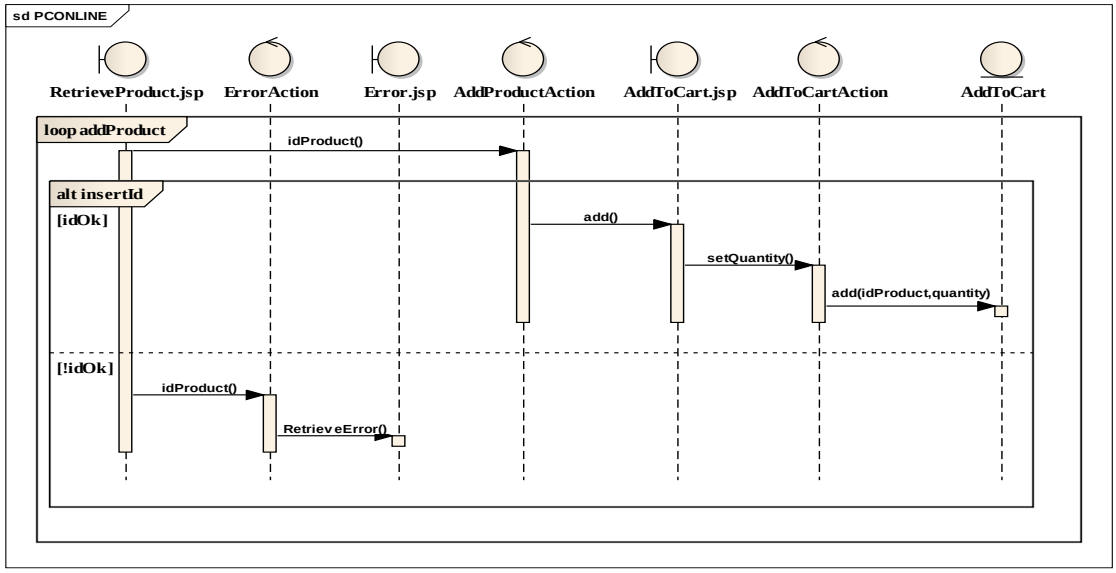

Figure 7. The Sequence Diagram of Add to Cart Use Case

The following section is dedicated for creating the EMF model and the XMI file of the sequence diagram cited above.

4.1. PIM Model of Add to Cart Use Case:

In this section, we present the PIM model of the sequence diagram of Add to Cart use case. Figure 8 shows this model.

```
<emf.model.sdModel:Sequencexmi:version="2.0"mlns:xmi="http://www.omg.org/
  XMI" xmlns:emf.model.sdModel="http:///emf/model/sdModel.ecore">
<lifelines type="RetrieveProduct.jsp">
<occs kind="Start" id="1"/>
<occs kind="fragm" id="2"/>
   <occs kind="event" id="3"msgs="//@messages.0"/>
<occs kind="fragm" id="4"/>
</lifelines>
<lifelines type="AddProductAction">
       <occs kind="Start" id="5"/>
   <occs kind="fragm" id="6"/>
  <occs kind="event" id="7"
  msgr="//@messages.0"/>
  <occs kind="fragm" id="8"/>
</lifelines>
<lifelines type="ErrorAction">
<occs kind="Start" id="9"/>
<occs kind="fragm" id="10"/>
</lifelines>
<lifelines type="Error.jsp">
<occs kind="Start" id="11"/>
<occs kind="fragm" id="12"/>
</lifelines>
<lifelines type="AddToCart.jsp">
<occs kind="Start" id="13"/>
<occs kind="fragm" id="14"/>
</lifelines>
<lifelines type="AddToCartAction">
<occs kind="Start" id="15"/>
<occs kind="fragm" id="16"/>
</lifelines>
<lifelines type="AddToCart">
<occs kind="Start" id="17"/>
<occs kind="fragm" id="18"/>
</lifelines>
<messages signal="idProduct()"
   send=//@lifelines.0/@occs.2receive="//@lifelines.1/@occs.2"/>
<messages signal="idProduct()" send= "//@fragm.1/@operand.1/@ll.0/@occs.1"
   receive="//@fragm.1/@operand.1/@ll.1/@occs.1"/>
<messages signal="RetrieveError()" send="//@fragm.1/@operand.1/@ll.1/
   @occs.1" receive="//@fragm.1/@operand.1 /@ll.2/@occs.1"/>
<messages signal="add()" send=//@fragm.1/@operand.0/@ll.0/@occs.1
   receive="//@fragm.1/@operand.0/@ll.1/@occs.1"/>
<messages signal="setQuantity()"
   send=//@fragm.1/@operand.0/@ll.1/@occs.1
   receive="//@fragm.1/@operand.0/@ll.2/@occs.1"/>
<messages signal="add(idProduct,quantity)"
   send="//@fragm.1/@operand.0/@ll.2/@occs.1"receive="//@fragm.1/@operand.0
   /@ll.3/@occs.1"/>
<fragm type="addProduct">
<fragm kind="fragm" id="2"/>
<fragm kind="fragm" id="6"/>
<fragm kind="fragm" id="10"/>
<fragm kind="fragm" id="12"/>
<fragm kind="fragm" id="14"/>
<fragm kind="fragm" id="16"/>
<fragm kind="fragm" id="18"/>
</fragm>
<fragm type="InsertId">
<fragm kind="fragm" id="4"/>
<fragm kind="fragm" id="8"/>
<operand guard="[idOk]">
<ll type="AddProductAction" >
```

```
<occs kind="start" id="19"/>
<occs kind="event" id="20" msgs="//@messages.3"/>
<occs kind="end" id="21"/>
</ll>
<ll type="AddToCart.jsp">
<occs kind="start" id="22"/>
   <occs kind="event" id="23" msgs="//@messages.4" msgr="//@messages.3"/>
<occs kind="end" id="24"/>
</ll>
<ll type="AddToCartAction">
<occs kind="start" id="25"/>
   <occs kind="event" id="26" msgs="//@messages.5" msgr="//@messages.4"/>
<occs kind="end" id="27"/>
</ll>
<operand guard="[!idOk]">
<ll type="RetrieveProduct.jsp">
<occs kind="start" id="31"/>
   <occs kind="event" id="32" msgs="//@messages.1"/>
<occs kind="end" id="33"/>
</ll>
<ll type="ErrorAction">
<occs kind="start" id="34"/>
   <occs kind="event" id="35" msgs="//@messages.2" msgr="//@messages.1"/>
<occs kind="end" id="36"/>
</ll>
<ll type="Error.jsp">
<occs kind="start" id="37"/>
   <occs kind="event" id="38" msgr="//@messages.2"/>
<occs kind="end" id="39"/>
```

Figure 8. The XMI File of Add to Cart Use Case

5. Transformation Rules Written in ATL:

The model transformation plays a role in the model driven engineering. To this end, several studies have been conducted to define transformation languages effectively and to ensure traceability between the different types of MDA models. ATL (Atlas Transformation Language) is a model transformation language developed in the framework of the ATLAS project [4-6]. ATL is developed by the team of Jean Bézivin at LINA in Nantes. It is part of the Eclipse M2M (Model-to-Model). The figure 9 shows the operational framework of ATL transformation.

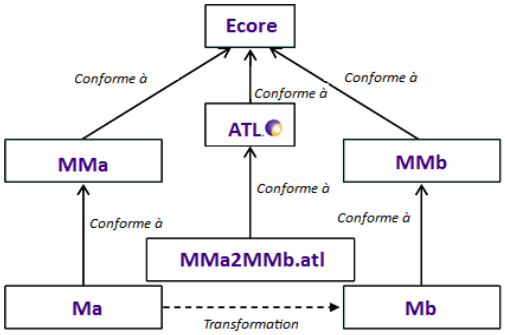

Figure 9. Operational Framework of ATL

The ultimate goal of the presented paper is to automate a Model-to-Model Transformation from the user domain model, which yields an intermediate model. In this paper, we write the

transformation rules in ATL transformation language. The result of these rules is a XMI file. The generated model is composed of three packages. In this section, we develop the ATL transformation rules and the packages generated by these rules. The Figures 10-17 show the different rules and the package generated by each rule.

5.1. Rules Specification:

Here are the main rules to transform the combination of CD and SD models into an MVC 2 Web model:

- The View Package is composed of a set of JSP pages.
- The Controller Package is composed of a set of mappings Action.
- The Controller Package is composed of a set of bean Forms.
- The mapping Action is composed of a set of Actions.
- The Bean Form is composed of a set of Action Form.
- Each operation generates an Action Form.
- The Retrieve operation of the root class does not generate an Action Form class.
- Each Operation generates an Action Class, an Action Form class and a Jsp Page.
- Each operation which is the subject of a transformation must belong to the sequence diagram model and the class diagram model.
- Each Action is composed of a set of *Action Forward* classes.
- An update operation generates an *Action Form* and *Action Form End* classes.

5.2. ATL Code:

The ATL code for the combination of CD and SD to MVC 2 Web model transformation consists in 5 rules. The Figures 10, 12 and 14 show these rules.

- The *UMLPackage2ViewPackage* rule generates the view package of jsp pages.
- The *Operation2JspPage* rule creates the jsp pages from an operation. The name of jsp page is the name of the operation.
- The *UMLPackage2ControllerPackage* rule generates the controller package.
- The *Operation2Action* rule creates the Action classes and the jsp page forwarding of each Action class.
- The *Operation2ActionForm* rule creates the *Action Form* classes.

A. Rule 1: From Operation to Jsp Pages

```
rule P2View{
 from
        a : UML!UML
 to
        vout : STRUTS!ViewPackage(
          name <- a.name,
          view <- Sequence{thisModule.allMethodDefs
                 ->collect(e |  thisModule.resolveTemp(e, 'jsp'))}
                )
             }

rule O2JspPage{

      from
      c : UML!Operation
      to
       jsp : STRUTS!JspPage (
                  name <- if c.name='Delete'  then
                            OclUndefined
                          Else   c.name+c.class.name+'.jsp'
                         endif
                         )
                }
```

Figure 10. The Rule that Generates the Jsp Pages Package

This rule can generate a package that contains the different jsp pages which is used in the user interface. The generated package is as follow:

```
<ViewPackage>
<view name="AddToCart.jsp"/>
<view name="RetrieveProduct.jsp"/>
</ViewPackage>
```

Figure 11. The Generated Result: jsp Pages Package

B. Rule 2: From Operation to Action Form

```
rule O2ActionForm{

    from
        c : UML!Operation
    to

        actf : STRUTS!ActionForm (
            name <- if  c.class.opposite.type.name='Void'
                    then
                     if c.name='Retrieve'
                        then OclUndefined
                     else  c.name+c.class.name+'Form'
                     endif
                else c.name+c.class.name+'Form'
              endif
            ),

        actf1 : STRUTS!ActionForm (
            name <- if c.name='Create'
                        then  c.name+c.class.name+'End'+'Form'
                   else if  c.name='Update'
                            then  c.name+c.class.name+'End'+'Form'
                       else OclUndefined
                       endif
                  endif
                )
        }
```

Figure 12. The Rule that Generates the Package of Action Form Classes

The above rule can generate a package that contains the different Action Form classes. The generated package is as follow:

```
<formbeans name="form-beans">
        <form name="AddProductForm"/>
        <form name="AddToCartForm"/>
</formbeans>
```

Figure 13. The Generated Result: Package of Action Form Classes

C. Rule 3: From Operation to Action

```
rule UML2ActionMapping{
    from
        a : UML!UML
    to
        act : STRUTS!ActionMapping(
          name <- 'action'+'-'+'mappings',
          action <- Sequence{thisModule.allMethodDefs
        ->collect(e | thisModule.resolveTemp(e, 'frm')}
          )
      }
rule O2Action{

      from
          c : UML!Operation
      to
           frm : STRUTS!Action(

                  path <- '/'+c.name+c.class.name,
                  name<- if c.class.opposite.type.name='Void'
                              then if c.name='Retrieve'
                   then OclUndefined
                          else  c.name+c.class.name+'Form'
                endif
              else c.name+c.class.name+'Form'
          endif,
            type <- c.name+c.class.name+'Action',
            input <- if c.class.opposite.type.name='Void'
                        then OclUndefined
                  else if c.name='Delete'
                        then   /'+'Retrieve'+c.class.opposite.type.name+'.jsp'
                     else    '/'+c.name+c.class.opposite.type.name+'.jsp'
                  endif
           endif,
          forward <- Sequence{fr}
          ),

     fr : STRUTS!ActionForward(
          name <- 'Success',
          path <- jsp
               )
}
```

Figure 14. The Rule that Generates the Package of Action Classes

In This rule, we can generate a package that contains the different Action classes. The generated package is as follow:

```
<actionmappings>
<action path="/AddProduct" name="AddProductForm"
     type="/AddProductAction" input="/RetrieveProduct.jsp">
<forward name="Success" path="/0/@view.0"/>
</action>
<action path="/AddToCart" name="AddToCartForm"
type="/AddToCartAction" input="/RetrieveProduct.jsp">
<forward name="Success" path="/0/@view.0"/>
</action>
</actionmappings>
```

Figure 15. The Generated Package of Action Classes

D. The PSM Model Generated:

In this section, we present the PSM model generated by the ATL transformation and its equivalent in EMF. Figure 16 shows the EMF model and Figure 17 shows the PSM model generated by ATL transformation rules.

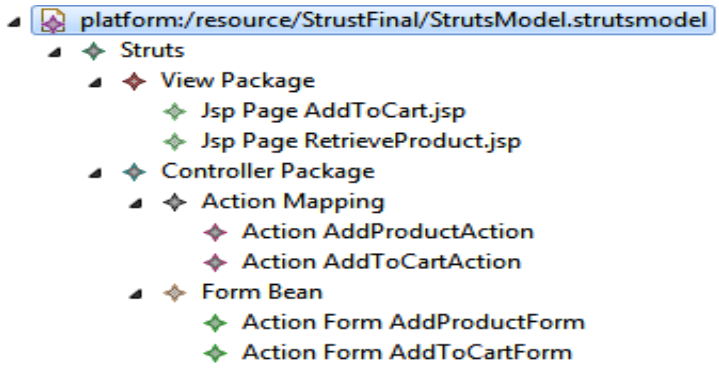

Figure 16. The PSM Model Generated

The different rules presented above, can generate an MVC2 Web model. This model contains the different packages necessary of an MVC2 webapplication. Figure 17 shows this model.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="STRUTS">
  <ViewPackage name="strutsModel">
    <view name="AddToCart.jsp"/>
    <view name="RetrieveProduct.jsp"/>
  <view/>
  </ViewPackage>
  <ControllerPackage>
    <actionmappings>
      <action path="/AddProduct" name="AddProductForm" type="/AddProductAction" input="/RetrieveProduct.jsp">
        <forward name="Success" path="/0/@view.0"/>
      </action>
      <action path="/AddToCart" name="AddToCartForm" type="/AddToCartAction" input="/RetrieveProduct.jsp">
        <forward name="Success" path="/0/@view.0"/>
      </action>
   </actionmappings>
    <formbeans name="form-beans">
      <form name="AddProductForm"/>
      <form name="AddToCartForm"/>
    </formbeans>
  </ControllerPackage>
</xmi:XMI>
```

Figure 17. The Target Model Generated by ATL Transformation Rules

6. The Code Generation Process:

Code generation isn't a new concept. It's been around for a while and has been gaining popularity with the model-driven development (MDD) movement as a way to increase productivity. The Eclipse project has a technology project called JET (Java Emitter Template) [7] that is a specialized code generator.

6.1. Java Emitter Template (JET)

JET is a "model to text" (M2T) engine which allows generating (text) output based on an EMF model. For example you can generate SQL, Java, XML, Text, HTML, *etc.*, JET uses a template technology which is very closely related to the Syntax of Java Server Pages (JSPs).

In JET we define templates. These templates will be used to create Java Implementation classes. This process step is called translation. .

The Java classes can then be used to create the final output, *e.g.*, a HTML file. This generated class can be initialized and will create the desired result as a String with the method "generate()". This process step is called generation.

JET has three different types of expressions, e.g. directives, expressions and scrip lets. Scrip lets are started with <% and ended with %> and can contain any java code. Expressions allow insert string values within the JET output and the directives define the settings for the JET template.

The JET compiler creates a Java source file for each JET. The suggestion for the JET templates is to use the following naming schema: ClassName.outputsu$formbean/@name)}.javaffixjet, whereby the output suffix determines the output, *e.g.*, java for Java Source or html for HTML files.

6.2. JET Project

In this section we present the elements necessary to create a JET project and after the generation of the MVC2 web e-commerce code that is a PC online shopping. The figure 18 shows these templates.

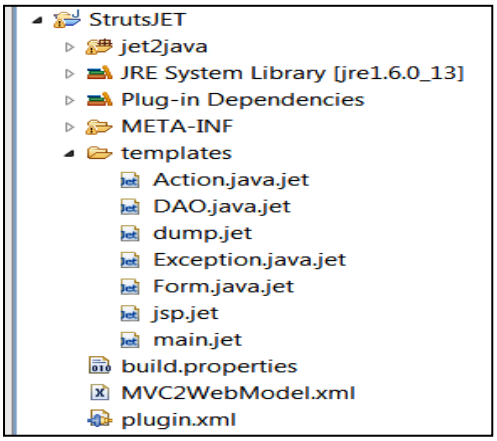

Figure 18. The Different Templates of the JET Project

From the Figure 18, we distinguish the following templates:

- *main.jet*: this serves an entry point to the transformation and generally invokes one or more templates that write content.
- *dump.jet* : which merely writes the input model to a file *dump.xml* in the JET project's root. The *dump.jet* template is invoked by *main.jet*.
- *Action.java.jet:* used to generate all Action classes.
- *Form.java.jet*: used to generate the Action Form classes.
- *DAO.java.jet:* used to generate the DAO classes. These classes contain the operations of each class.

- *jsp.jet:* template used to generate the jsp classes.

In this work, we develop thereafter the different templates cited below excluding the DAO template. The code of each template will be detailed as follows in the following sections:

6.2.1. The Main Template:

```
<%-- Main entry point for StrutsJET --%>
<ws:project name="StrutsSample" >
    <ws:folder path="src" >
        <c:iterate select="/MVC2WebModel/ControllerPackage/formbeans/form[@type='Class']" var="formbean">
            <java:package name="projet.struts.ActionForm">
                <ws:file path="{className($formbean/@name)}.java" template="templates/Form.java.jet"/>
            </java:package>
        </c:iterate>
        <c:iterate select="/MVC2WebModel/ControllerPackage/actionmappings/action" var="action" >
            <java:package name="projet.struts.Action" >
                <ws:file path="{className($action/@type)}.java" template="templates/Action.java.jet"/>
            </java:package>
        </c:iterate>
        <c:iterate select="/MVC2WebModel/DAO/dao" var="dao" >
            <java:package name="projet.struts.DAO">
                <ws:file path="{className($dao/@nameDAO)}.java" template="templates/DAO.java.jet"/>
            </java:package>
        </c:iterate>
    </ws:folder>
</ws:project>
<c:iterate select="/MVC2WebModel/ViewPackage/view" var="view" >
    <ws:file path="/StrutsGen/WebContent/jsp/{$view/@name}" template="templates/jsp.jet"/>
</c:iterate>
<%-- Copy the MVC2WebModel.xml into the WEB-INF directory --%>
<ws:project name="StrutsSample" >
    <ws:folder path="WebContent/WEB-INF" >
        <ws:file template="templates/dump.jet" path="MVC2WebModel.xml"/>
    </ws:folder>
</ws:project>
```

Figure 19. The Main Jet Template and the Numbers of Blocks

The main.jet template consists of a set of tags. Each tag is designed for a specific task. To generate a Java class, a text file or even XMI file in a fixed directory and also add a package file create a folder, a package of folder and a file of this package, we need to a block of tags. In this template, we have 5 blocks of tags. Each block is intended to generate a java class or generate a JSP page or a configuration file. The blocks are numbered from 1 to 5. The figure 20 shows the main.jet template and the blocks numbers.

This 1 will run the Form java.jet template on the input model and dump the results at {class Name(. This will create a collection of Action Form java classes.

The block 2 will run the Action.java.jet template on the input model and dump the results at {className($action/@type)}.java. This will create a collection of Action Java classes.

This 3 will run the DAO.java.jet template on the input model and dump the results at {className($dao/@type)}.java. This will create a DAO Java class. The DAO java class contains all methods that are the CRUD and the other methods.

This 4 will run jsp.jet template on the input model and dump the results at /StrutsGen/WebContent/jsp/{$view/@name}. This will create a set of jsp pages.

This 5 will copy the MVC2WebModel.xml into the WEB-INF directory.

6.2.2. The Action Class Template

In this section, we present the template of Action class. A Struts action is an instance of a subclass of an Action class, which implements a portion of a Web application and whose perform or execute method returns a forward [20]. Figure 20 shows this template.

```
// Autogenerated by JET at <f:formatNow pattern="MM/dd/yyyy HH:mm" />
public class <c:get select="className($action/@type)"/> extends Action {

    public ActionForward execute(final ActionMapping pMapping,
            ActionForm pForm, final HttpServletRequest pRequest,
            final HttpServletResponse pResponse) {

        final DynaValidatorActionForm lForm = (DynaValidatorActionForm)pForm;

        <c:iterate select="/MVC2WebModel/ControllerPackage/formbeans/form[@name=className($action/@name)]/formp" var="formb">
            final <c:get select="className($formb/@type)" /> l<c:get select="className($formb/@name)"/> =
                                                        lForm.getString("<c:get select="className($formb/@name)"/>");
        </c:iterate>
        // Create anew record
        <c:iterate select="/MVC2WebModel/DAO/dao[@nameDAO=className($action/@nameDAO)]" var="dao">

            final <c:get select="className($action/@nameDAO)"/> l<c:get select="className($action/@nameDAO)"/> =
                                                    new <c:get select="className($action/@nameDAO)"/>();
final String lError =
    l<c:get select="className($action/@nameDAO)"/>.<c:get select="className($action/@type)"/>(<c:get select="className($action/@nameDAO)"/>,);
        </c:iterate>
        if(lError == null) {
        return pMapping.findForward("succes");
        }
        else {
            final ActionMessages lErrors = getErrors(pRequest);
            final ActionMessage lActionMessage = new ActionMessage(lError, false);
            lErrors.add(Globals.ERROR_KEY, lActionMessage);
            saveErrors(pRequest, lErrors);
        return pMapping.findForward("error");
        }
    }
}
```

Figure 20. Action Class Template

6.2.3. The Action FormTemplate

In this section, we developed the Action Form class template. Figure 22 shows this template.

```
<\%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<\%@ taglib uri="/WEB-INF/tld/struts-html.tld" prefix="html" %>
<\%@ taglib uri="/WEB-INF/tld/struts-logic.tld" prefix="logic" %>
<\%@ taglib uri="/WEB-INF/tld/struts-bean.tld" prefix="bean" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
    <head>
        <title><bean:message key="title.add.<c:get select="className($action/@type)"/>"" bundle="add"/></title>
    </head>
    <body>
        <b><i><html:errors/></i></b><br/>
        <html:form action="/<c:get select="className($action/@type)"/>.do">
            <bean:message key="add.order.libelle.libelle" bundle="add"/>
            <nested:text property="id<c:get select="className($action/@type)"/>"/><br>
            <html:submit/>
        </html:form>
        <table border="1">
            <thead>

                <tr>
                <c:iterate select="/MVC2WebModel/DAO/dao[@nameDAO=className($action/@nameDAO)]" var="dao">
                    <th><bean:message key="column.<c:get select="className($formb/@name)"/>"/></th>
                </c:iterate>
                </tr>
            </thead>
            <tbody>
                <logic:iterate id="element<c:get select="className($action/@type)"/>" name="LISTE_<c:get select="className($action/@type)"/>S"
                                                        type="ecommerce.project.Bean.<c:get select="className($action/@type)"/>Bean">
                <tr>
                <c:iterate select="/MVC2WebModel/ControllerPackage/formbeans/form[@type='Class']" var="formb">
                  <td>
                    <bean:write name="element<c:get select="className($action/@type)"/>" property="<c:get select="className($formb/@name)"/>"/>
                  </td>
                </c:iterate>
                </tr>
                </logic:iterate>
            </tbody>
        </table>
    </body>
</html:html>
```

Figure 21. Action Form Template

```
// Autogenerated by JET at <f:formatNow pattern="MM/dd/yyyy HH:mm" />
public class <c:get select="className($formbean/@name)" /> extends ActionForm {

      <c:userRegion>
           // BEGIN REGION RESET
           <c:initialCode unmodifiedMarker="@generated" >
           // @generated
             super.reset(mapping, request);
           </c:initialCode>
           // END REGION RESET
       </c:userRegion>

   private static final long serialVersionUID = -8922507342446865665L;

   private static Upd<c:get select="className($formbean/@nameClass)"/>Factory FACTORY_INSTANCE =
                                                  new Upd<c:get select="className($formbean/@nameClass)"/>Factory();

   private static class Upd<c:get select="className($formbean/@nameClass)"/>Factory implements Factory {
       public Object create() {
           return new Upd<c:get select="className($formbean/@nameClass)"/>Bean();
       }
   }

   @Override
   public void reset(ActionMapping pMapping, HttpServletRequest pRequest) {
       super.reset(pMapping, pRequest);

       final List<Upd<c:get select="className($formbean/@nameClass)"/>Bean> lListUpdate =
                                                    new LinkedList<Upd<c:get select="className($formbean/@nameClass)"/>Bean>();
       final List<Upd<c:get select="className($formbean/@nameClass)"/>Bean> lListUpdateLazy =
                                                     LazyList.decorate(lListUpdate, FACTORY_INSTANCE);
       set("lists", lListUpdateLazy);
   }
   @SuppressWarnings("unchecked")
   @Override
  public ActionErrors validate(ActionMapping pMapping, HttpServletRequest pRequest) {
       final ActionErrors lActionErrors = super.validate(pMapping, pRequest);

 final List<Upd<c:get select="className($formbean/@nameClass)"/>Bean> lListUpdElementsDep =
               (List<Upd<c:get select="className($formbean/@nameClass)"/>Bean>)get("list<c:get select="className($formbean/@nameClass)"/>s");
       final Iterator<Upd<c:get select="className($formbean/@nameClass)"/>Bean> lIterators<c:get select="className($formbean/@nameClass)"/> =
                                                        lLists<c:get select="className($formbean/@nameClass)"/>.iterator();
while(lIterators<c:get select="className($formbean/@nameClass)"/>.hasNext()) {
  final Upd<c:get select="className($formbean/@nameClass)"/>Bean lElement<c:get select="className($formbean/@nameClass)"/> =
                                                        lIterators<c:get select="className($formbean/@nameClass)"/>.next();
   <c:iterate select="/formbeans/form/formp" var="p" >

       final String l<c:get select="$p/@name" /> = lElement<c:get select="className($formbean/@nameClass)"/>.get<c:get select="$p/@name" />();
   </c:iterate>

       }
   return lActionErrors;
   }
 }
}
```

Figure 22. Template of jsp Page

6.2.4. The Jsp Page Template

In this section, we present the jsp pagetemplate. Figure 22 shows this template.

7. The Result of Code Generation Process:

In this section, we present the result of code generation process. This result is constituted from the Action classes, the Action Form classes and the jsp pages. We show an example of each generated elements. Figure 23 shows the Action class generated, the Figure 24 shows the Action Form class and the Figure 25 shows the jsp page.

7.1. The Action Class:

In this example, we present the Add To Cart Action class. This class is generated from the Action package generated in the PSM model.


```
public class AddToCartAction extends Action{

    public ActionForward execute(final ActionMapping pMapping,
            ActionForm pForm, final HttpServletRequest pRequest,
            final HttpServletResponse pResponse) {

        final DynaValidatorActionForm lForm = (DynaValidatorActionForm)pForm;
        final String lidCart = lForm.getString("idCart");
        final String lqty = lForm.getString("qty");
        final String lprice = lForm.getString("price");

        // Create anew record
        final listAddToCartDAO lListProductDAO = new listAddToCartDAO();
        final String lErreur = lListAddToCartDAO.addToCart(lidCart,lqty,lprice);
        if(lErreur == null) {
            return pMapping.findForward("succes");
        }
        else {
            final ActionMessages lErrors = getErrors(pRequest);
            final ActionMessage lActionMessage = new ActionMessage(lError, false);
            lErreurs.add(Globals.ERROR_KEY, lActionMessage);
            saveErrors(pRequest, lErrors);

            return pMapping.findForward("error");
        }
    }
}
```

Figure 23. The AddToCartAction.Java Class Generated

7.2. The Action Fom Class:

In this section we show the Action Form class generated. This class is generated from the Action Form package generated in the PSM model but the attributes are generated from the Property package.

```
public class AddToCartForm extends ActionForm {

    private static final long serialVersionUID = -8922507342446865665L;
    private static UpdAddToCartFactory FACTORY_INSTANCE = new UpdAddToCartFactory();

    private static class UpdAddToCartFactory implements Factory {
        public Object create() {
            return new UpdAddToCartBean();
        }
    }
    public void reset(ActionMapping pMapping, HttpServletRequest pRequest) {
        super.reset(pMapping, pRequest);
        final List<UpdAddToCartBean> lListUpdate = new LinkedList<UpdAddToCartBean>();
        final List<UpdAddToCartBean> lListUpdateLazy = LazyList.decorate(lListUpdate, FACTORY_INSTANCE);
        set("listUpdAddToCarts", lListUpdateLazy);
    }
    @SuppressWarnings("unchecked")
    @Override
    public ActionErrors validate(ActionMapping pMapping, HttpServletRequest pRequest) {
        final ActionErrors lActionErrors = super.validate(pMapping, pRequest);
        final List<UpdAddToCartBean> lListUpdElementsAddToCart = (List<UpdAddToCartBean>)get("listAddToCarts");
        final Iterator<UpdAddToCartBean> lIteratorUpdElementsAddToCart = lListUpdlementsAddToCart.iterator();
        while(lIteratorUpdElementsAddToCart.hasNext()) {
            final UpdAddToCartBean lElementAddToCart = lIteratorUpdElementsAddToCart.next();
            final Long lidCart = lElementUpdAddToCart.getIdCart();
            final String lqty = lElementUpdAddToCart.getQty();
            final Long lprice = lElementUpdAddToCart.getPrice();
        }
        return lActionErrors;
    }
}
```

Figure 24. The AddToCartForm.Java Class Generated

7.3. The Jsp pages:

In this section, we show the code generated for the AddToCart jsp page. This jsp page is generated from the Jsp Page package generated in the PSM model. This jsp page is as follows:

```
<%@ taglib prefix="html" uri="http://struts.apache.org/tags-html" %>
<%@ taglib prefix="bean" uri="http://struts.apache.org/tags-bean" %>
<%@ taglib prefix="logic" uri="http://struts.apache.org/tags-logic" %>
<%@ taglib prefix="nested" uri="http://struts.apache.org/tags-nested" %>
<html:html>
    <head>
        <title><bean:message key="titre.creation.cart" bundle="add"/></title>
    </head>
    <body>
        <b><i><html:errors/></i></b><br/>
        <html:form action="/CreateCreationCart.do">
            <bean:message key="creation.order.libelle.libelle" bundle="creation"/>
            <nested:text property="idCart"/><br>
            <html:submit/>
        </html:form>
        <table border="1">
            <thead>
                <tr>
                    <th><bean:message key="column.idCart"/></th>
                    <th><bean:message key="column.qty"/></th>
                    <th><bean:message key="column.price"/></th>
                </tr>
            </thead>
            <tbody>
                <logic:iterate id="elementCart" name="LISTE_CARTS" type="com.project.struts.Bean.AddToCart">
                    <tr>
                        <td><bean:write name="elementCart" property="idCart"/></td>
                        <td><bean:write name="elementCart" property="qty"/></td>
                        <td><bean:write name="elementCart" property="price"/></td>
                    </tr>
                </logic:iterate>
            </tbody>
        </table>
    </body>
</html:html>
```

Figure 25. The Jsp Page Generated: AddToCart.jsp

8. Evaluation

After generating the code, we want to know the percentage of the generated code with respect to the total application code. In this case study, we obtained a very respectful percentage of code generation occurred. The generated code has reached about 85% of total code.

In the next work, we want to ameliorate this application of code generation in order to achieve a 100% of the code.

9. Related Work

From a state-of-the-art review in model transformation and code generation, several proposals have been identified. The most relevant are [10-18, 21].

In [10] Cooper, *et al.*, allows code generation from models to aspects in AspectJ, a java implementation of aspect-oriented programming (AOP). The transformation among models is accomplished by means of Extensible Markup Language (XML) specifications and meta-models of XML and AspectJ. The code is generated from the XML specifications and the aspects are controlled in the system by throwing and handling exceptions.

Nassar, *et al.*, [11] propose a method for code generation by merging use-case-based view point models, logic, component, and deployment. The models are stereotyped according to elements of VUML (View-based Unified Modeling Language) and, then, transformed into code by using predefined rules specified in the ATLAS Transformation Language (ATL). The aim of the generated code is to manage the different views of the system, excluding business logic.

A relevant work on the code generation from formal representations was conducted. This is the case of PADL2 Java [12], which defines transformation rules to generate code in Java from specifications in an algebraic formal language called PADL. The system is completely specified on PADL, easing the process for generating both structural and behavioral codes.

Feiler, *et al.,* [13] use a custom UML profile for modeling PIMs using state machine diagrams and a proprietary language for actions. Afterwards, they transform PIM into PSM based on AADL using ATL (Atlas Transformation Language). From AADL, code is generated containing structure and behavior. Chehade, *et al.,* [14] show one way to reduce the cost of portability in MDA by providing domain-independent model transformations. Lin, *et al.,* [15] present a framework and show how software code can be automatically generated from SysML models of multi-core embedded systems.

Some researchers have opted for a combination of models and textual specifications, and the benefits of both. Fang [16] combines modeling patterns and action semantic with MDA to create applications for a specific platform, called EJB. Patterns and platform are expressed with UML and action semantic representations. The latter is based on the UML meta-model. Although the use of meta-models could allow for the extension of the method to other platforms, the strong link between patterns and the platform could make this step difficult. Also, in some cases the code obtained from patterns is structural and not behavioral. Sánchez, *et al.,* [17] shows a graphic DSL for the home automation domain. The structure of the DSL is composed of so-called functional units, common functionalities on the domain (*e.g.*, light power on/off or lighting level regulation associated to services like dimmers and timers). A program with DSL consists of a sequence of services and its actions. The code generated is fully executable due to the defined behavior exhibited by every functional unit, which acts as a code template. The DSL has an internal XML representation, the main source of the transformation rules. The proposal is domain-platform independent because its structure is flexible enough to allow for the implementation of new services and functional units in several devices and platforms.

On the other hand, Muñetón, *et al.,* [18] shows a novel approach for code generation which combines UML graphs and semantic annotations. This improvement allows for one to generate structural and behavioral codes and uses a DSL for making the annotations easy enough to learn and understand for developers, designers, and business stakeholders.

In [21] the author combines the UML class diagram and UML activity diagram for generating the code of an e-commerce web application. The objective of using an activity diagram is to establish the UML class diagram and know the input jsp of each Action class.

In this paper, we combine CD and SD diagrams in order to generate precisely the ingredients of MVC 2 web. The combination of these diagrams allows to specify and to control the different input jsp pages, so we can determine precisely the different Action classes, the jsp pages associated with these classes, its Action Form classes and the attributes of each Action Form class.

10. Conclusion and Future Work:

This work approaches the problem of developing web-based systems by utilizing MDA based on model transformation. In particular, we combine the UML class diagram and UML sequence diagram and thereafter we transform the result model of this combination into a framework based on MVC design patterns (Struts framework). In this transformation, we generate the necessary classes that implement the model and controller and the jsp pages that implement the view respecting the specificities of the MVC 2 pattern.

The aim of use the sequence diagram is to illustrate the interaction of objects within a scenario of a use case diagram. The goal is to describe how the actions occur between the actors or objects. In this work, we use the sequence diagram to know the link between the *Action* classes, the *Action Form* classes and the *jsp* pages. Through this sequence diagram, we can especially know the input *jsp* page of another page which it will be need. Another

advantage of this diagram is to define *Action Form* attributes and the object or class which each attribute belongs to.

The use of UML class diagram gives a general idea about the link between the classes, methods and attributes of each class as well as associations between classes. The combination of UML class diagram and UML sequence diagram is to generate precisely the elements that constitute the ingredients ofMVC2 Web and the relations between these elements.

This work has described the process of transforming an UML class diagram combined with a sequence diagram into a deployable e-commerce Web application. This process is divided into two steps. The first is the M2M transformation and the second is the M2T transformation.

Furthermore, we plan to generate an e-commerce web code from the integration of various frameworks like Struts 2, Spring and Hibernate. Our goal is to facilitate more and more the development and tests.

Acknowledgement

The author(s) declare(s) that there is no conflict of interests regarding the publication of this paper.

References

[1] "OMG Model Driven Architecture", document update/2012-06-27, **(2013),** (http://www.omg.org/mda/).
[2] X. Blanc, "MDA en action: Ingénierie logicielle guidée par les modèles", Eyrolles, **(2005).**
[3] S. J. Mellor and K. Scott, A. Uhl, D.Weise, "MDA Distilled: Principles of Model-Driven Architecture", Addison-Wesley, **(2004).**
[4] L. MENET, "Formalisation d'une Approche d'Ingénierie Dirigée par les Modèles Appliquée au Domaine de la Gestion des Données de Référence", PhDthesis, Université de Paris VIII, Laboratoire d'Informatique Avancée de Saint-Denis (LIASD), école doctorale: Cognition Langage Interaction (CLI), **(2010).**
[5] F. Jouault, F. Allilaire, J. Bézivin and I. Kurtev, "ATL : A Model Transformation Tool". Sciences of Computer Programming-Elseiver, vol. 72, no. 1-2, **(2008)**, pp. 31–39.
[6] "Omg/mof Meta Object Facility (MOF) specification", omg document ad/2012-06-27, **(2014),** (http://www.omg.org).
[7] **(2014)** March 10, http://www.eclipse.org/articles/Article-JET2/ jet_tutorial2.html.
[8] S. Mbarki and M. Erramdani, "Model-Driven Transformations: From Analysis to MVC 2 Web Model", International Review on Computers and Software (I.RE.CO.S), vol. 4, no. 5, **(2009)**.
[9] "OMG Unified Modelling Language (OMG UML)", Infrastructure OMG Document Number: formal/2010-05-03, **(2013),** http://www.omg.org/spec/UML/2.3/Infrastructure.
[10] J. Bennett, K. Cooper and L. Dai, "Aspect-Oriented Model-Driven Skeleton Code Generation: A Graph-Based Transformation Approach", Science of Computer Programming-Elseiver. vol. 75, no. 8, **(2010)**, pp. 689-725.
[11] M. Nassar, A. Anwar, S. Ebersold, B. Elasri, B. Coulette and A. Kriouile, "Code Generation in VUML Profile: A Model Driven Approach". IEEE/ACS International Conference on Computer Systems and Applications, **(2009)**, pp. 412 – 419.
[12] E. Bontà and M. Bernardo, "PADL2Java: A Java Code Generator for Process Algebraic Architectural Descriptions", IEEE/ACS International Conference on Computer Systems and Applications, **(2009)**, pp. 412 – 419.
[13] P. Feiler, D. Niz, C. Raistrick and B. Lewis, "From PIMs to PSMs", Proceedings of the 23rd IEEE International Conference on Engineering Complex Computer Systems, **(2007)**, Auckland, New Zealand.
[14] W. E. H. Chehade, A. Radermacher, F. Terrier, B. Selicy and S. Gerard, "A Model-Driven Framework for the Development of Portable Real-time Embedded Systems", Proceedings in the 16th IEEE International Conference on Engineering of Complex Computer Systems, **(2011)**.
[15] C.-S. Lin, C.-H. Lu, S.-W. Lin, Y.-R. Chen and P.-A. Hsiung, "VERTAF/Multi-Core: A SysML-Based Application Framework for Multi-Core Embedded Software Development". Journal of Computer Science and Technology, vol. 26, no. 3, **(2011)**, pp. 448–462, DOI 0.1007/s11390-011-1146-3.
[16] F. Cheng, "MDA Implementation Based on Patterns and Action Semantics", Third International Conference on Information and Computing, **(2010)**, pp. 25-28.

[17] P. Sánchez, M. Jiménez, F. Rosique, B. Á lvarez and A. Iborra, "A Framework for Developing Home Automation Systems: From Requirements to Code", The Journal of Systems and Software, vol. 84, **(2011)**, pp. 1008–1021.
[18] A. Muñetón and C. Zapata, "Definition of a Semantic Platform for Automated Code Generation Based on UML Class Diagrams and DSL Semantic Annotations", Dyna Journal, vol. 79, no. 172, **(2010)**, pp. 94-100, ISSN 0012-7353.
[19] R. Grønmo, F. Sørensen, B. Møller-Pedersen and S. Krogdahl, "A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol", In European Conference on Model Driven Architecture - Foundations and Applications (ECMDA)-Springer, **(2008)**, doi:10.1007/978-3-540-69100-6-18.
[20] **(2014),** http://struts.apache.org.
[21] M. Rahmouni and S. Mbarki, "An end-to-end Code Generation from UML Diagrams to MVC2 Web Applications", International Review on Computers and Software (I.RE.CO.S), vol. 8, no. 9, **(2013)**, pp. 2123-2135.

Authors

M'hamed Rahmouni, Ph. D Student, holding a Diploma of Higher Approfondie Studies in Computer Science and Telecommunication from the faculty of science, Ibn Tofail University, Morocco, 2007. He participated in various international congresses in MDA (Model Driven Architecture) integrating new technologies XML, EJB, MVC, Web Services, *etc.*

Samir Mbarki, he Received the B.S. degree in applied mathematics from Mohammed V University, Morocco, 1992, and Doctorat of High Graduate Studies degrees in Computer Sciences from Mohammed V University, Morocco, 1997. In 1995 he joined the faculty of science Ibn Tofail University, Morocco where he is currently a Professor in Department of computer science. His research interests include software engineering, model driven architecture, software metrics and software tests. He obtained an HDR in computer Science from Ibn Tofail University in 2010.