Laboratoire de Recherches sur les Technologies du Commerce Électronique

Université du Québec à Montréal

Rapport de recherche Latece 2017-3

Identifying KDM Model of JSP Pages

Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Ghizlane El Boussaidi, Jean Privat, Yann-Gaël Guéhéneuc, Naouel Moha

LATECE Laboratoire, Université du Québec à Montréal, Canada

July, 2017

Identifying KDM Model of JSP Pages

Anas Shatnawi[1], Hafedh Mili[2], Manel Abdellatif, Ghizlane El Boussaidi, Jean Privat, Yann-Gaël Guéhéneuc, Naouel Moha

LATECE Laboratoire, Université du Québec à Montréal, Canada

Abstract

In this report, we propose our approach that identifies a KDM model of JSP pages. Our approach is based on two main steps. The first one aims to reduce the problem space by translating JSP pages into Java Servlets where we can use existing tools to identify a KDM model. The second step aims to complete the resulting KDM model by identifying dependencies of JSP tags that are not codified by the translation step.

1 Introduction

J2EE applications are implemented based on components developed using a variety of technologies. These technologies are written following Java code (e.g., JavaBeans, Managed Beans, etc.) or scripting languages using XML tags (e.g., JSP, JSF).

To best to our knowledge, there is no approach/tool that identifies a KDM model for the JEE technologies implementing using scripting languages (HTML, JSP, JSF, ASP, etc.).

Therefore, in this paper, we propose an approach that identifies a KDM model of JSP pages. Our approach is based on two main steps. The first one aims to reduce the problem space by translating JSP pages into Java Servlets where we can use existing tools to identify a KDM model. To do so, we use *Jasper* tool for translating JSP to Java and the *MoDisco* tool for extracting KDM from Java. The second step aims to complete the resulting KDM model by identifying dependencies of JSP tags that are not codified by the translation process. We build a table of the set of the tags and their related attributes and developed a set of tools that parse JSP pages and configuration files to identify dependencies related to these tags.

The rest of this report is organized as follows. We provide an overview of the proposed approach in Section 2. Then, we show how we reduce the problem space by translating JSP pages to Java Servlets in Section 3. Next, we discuss the codification of the JSP tags in Section 4. We finish by concluding the report in Section 5.

2 Overview of the Proposed Approach

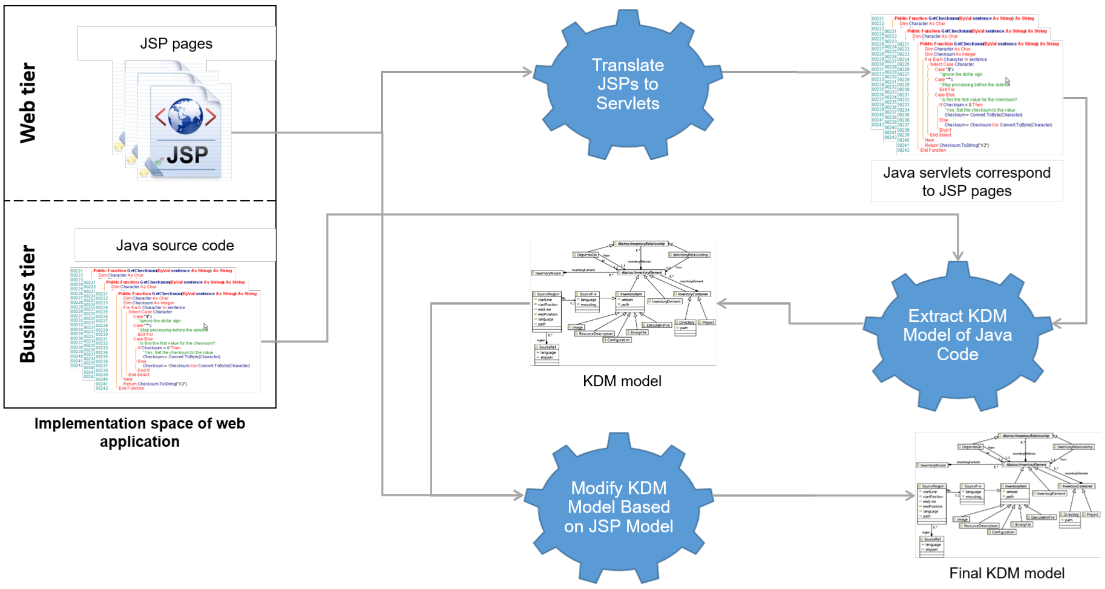
The gaol is to represent the implementation of JSP pages in a KDM model. This includes both the program elements and their related dependencies. The process of the proposed approach is based on two main steps:

[1]anasshatnawi@gmail.com
[2]mili.hafedh@uqam.ca

Step 1: The Reduction of the Problem Space The main idea behind this step is to map our problem to another already solved problem. Thanks to the MoDisco tool which supports the transformation of the normal Java code to KDM models. Therefore, we decide to translate JSP pages to equivalent Java code.

Step 2: The Codification of Dependencies Related to JSP Tags In the first step, we solve a part of the problem by translating the JSP implementation to Java code. However, there are some JSP tags that are not translated to Java code. Therefore, we aim, in this step, to identify these tags and codify their related dependencies in the KDM model.

The main process of our approach is shown in Figure 1.



Web tier
JSP pages
JSP
Business tier
Java source code
Implementation space of web application
Translate JSPs to Servlets
Java servlets correspond to JSP pages
Extract KDM Model of Java Code
KDM model
Modify KDM Model Based on JSP Model
Final KDM model

Figure 1: Process of identifying KDM model of JSP pages

3 The Reduction of the Problem Space

We select to translate the implementation of JSPs to Servlets (Java code) due to:

1. Servlets represent the underlying Java implementation of server pages. Thus, they support the life cycle management of JSP pages,

2. the availability of an open-source tool that translates JSP pages into Servlets such that the resulting Servlets exactly provide the same functionalities (output) of the corresponding JSP pages, and

3. the availability of an open-source tool that identifies KDM models of Servlets.

In this section, we present how to translate JSPs to Servlets using the Jasper tool. Then, we show how to identify a KDM model of these resulting Servlets using the MoDisco tool.

3.1 The Translation of JSPs to Servlets Using Jasper

To translate JSP pages to Servlet classes, we use the *Jasper* tool that is provided as a part of the Apache Tomcat server [1]. The process used by Jasper to translate JSP pages into Servlets is presented in Figure 2.

The Java class of a resulting Servlet is structured in three methods following the JSP life cycle that is shown in Figure 3. These methods are *_jspInit(...)*, *_jspService(...)* and *_jspDestroy(...)* that are respectively used to initialize the Servlet, to serve the requests arrived to the Servlet from clients and to remove the resources.

In the following sub-sections, we present the rules that Jasper used to translate a JSP page to a Servlet, the implementation of Jasper based on ant project and an example of of a JSP page and its translated Servlet.

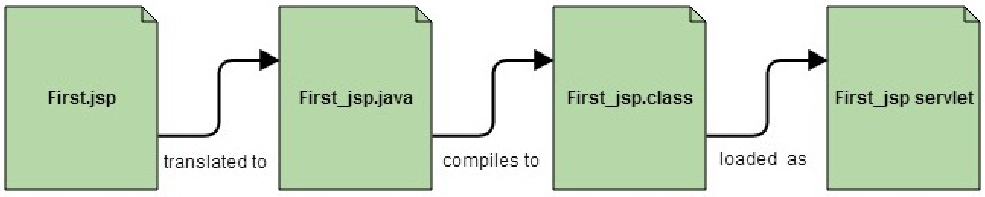


Figure 2: Process of converting JSP pages into Servlets by Jasper

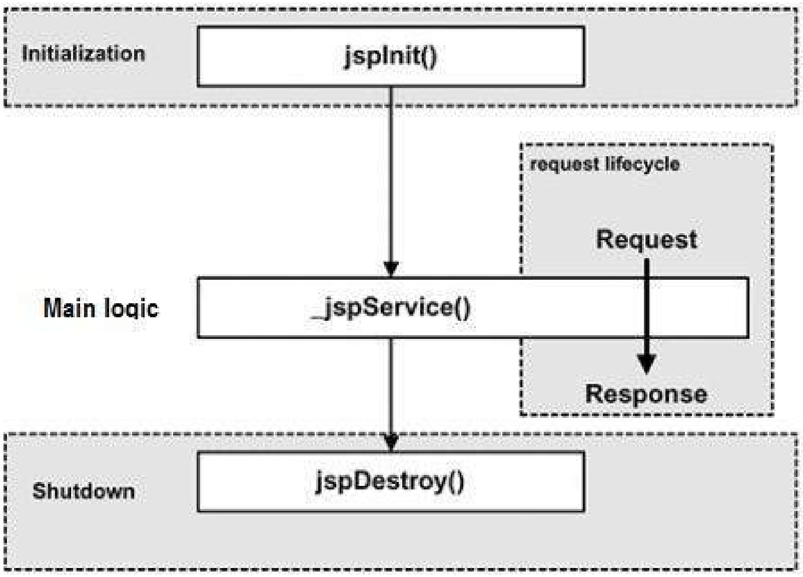


Figure 3: The life cycle of JSP pages

3.1.1 Transformation Rules of JSPs to Servlets Based on Jasper

The translation process of each JSP tag is done sequentially as it is implemented in the corresponding JSP page. The JSP tags is translated based on the following rules:

- **JSP scriptlet tags** (e.g., *<% code fragment %>*) that are used to insert Java code inside the JSP pages are represented in the Servlet class based on the same code fragment that it contains.

  For example, the *scriptlet <% for (int i=0; i<10; i++) %>* is translated to *for (int i=0; i<10; i++)*.

- **JSP declaration tags** (e.g., *<%! declaration; [declaration;]+ ... %>*) that are used for variable declarations are converted into equivalent variable declarations.

  For example, this declaration *<%! int i=0; !%>* is translated to *int i=0;*.

- **The reference to a JavaBeans component** is converted by creating an instance of the corresponding class. The references to setter/getter methods are realized through normal Java method invocations.

  For example, the *<jsp:useBean id="myBeans" class="package.BeansClass" scope="session">* tag is transformed into an object instantiation of *package.BeansClass*, and *<jsp:getProperty name="myBeans" property="firstName"> </jsp:getProperty>* into a method invocation to the *getFirstName()* method of this object instance.

- **The use of a custom tag handler** is realized in terms of a set of method invocations related to the life cycle management methods of the tag handler. Such methods are *doStartTag(), doEndTag() setAttribute()* etc.

- **Other JSP tags and HTML/XML tags** are only written as string literal parameters attached in the *out* object related to the response object (*out.write("...");*).

  For example, the *<jsp:include page="/myPage.jsp." flush="true" />* tag is converted as *out.write("<jsp:include page="/myPage.jsp." flush="true" />");*. These tags need further processing (c.f., Section 4).

A result of these transformation rules, we confirm that several program elements and their related dependencies of JSP pages are converted to Java code. Based on this Java code, we will identify a KDM model.

3.1.2 The Implementation of Jasper Tool Based on Ant Code to Translate JSPs to Servlets in a Web Application

To be able to convert JSPs into Servlets, the *Jasper* tool requires having the *Java SDK*[3] [2] and *Apache Ant*[5] installed on your machine.

In [3], Apache Tomcat provides the ant code that translates JSP pages of a given JEE Web application into Servlets based on Jasper. Listing 1 shows this code where one has to determine the directory of Apache Tomcat (i.e., *<property name="tomcat.home" ...*), the directory of the resulting Java code (*<property name="webapp.path"...*) and the directory related to the other output like the new generated *web.xml* and the *.class* compiled files.

[3]This video demonstrates how to install *Apache Tomcat*[4] [1], the Java SDK: https://www.youtube.com/watch?v=asoDE3AQXBA

[5]This video demonstrates how to install the Apache Ant: https://www.youtube.com/watch?v=39OwnrUlz0k

Listing 1: Ant code to convert JSP pages into Java servlets

```
<project name="Webapp Precompilation" default="all" basedir=".">

   <property name="tomcat.home" value="C:\Program Files\Apache Software Foundation\Tomcat 5.5"
       />
   <property name="webapp.path" value="C:\...ResultDirectory" />
   <property name="javacode.path" value="C:\...JavaCodeDirectory" />

   <target name="jspc">

      <taskdef classname="org.apache.jasper.JspC" name="jasper2">
      <classpath id="jspc.classpath">
         <pathelement location="${java.home}/../lib/tools.jar" />
         <fileset dir="${tomcat.home}/bin">
            <include name="*.jar" />
         </fileset>
         <fileset dir="${tomcat.home}/server/lib">
            <include name="*.jar" />
         </fileset>
         <fileset dir="${tomcat.home}/common/lib">
            <include name="*.jar" />
         </fileset>
      </classpath>
      </taskdef>

      <jasper2 validateXml="false" uriroot="${webapp.path}"
           webXmlFragment="${webapp.path}/WEB-INF/generated_web.xml"
           outputDir="${javacode.path}/jsps" />

   </target>

   <target name="compile">

      <mkdir dir="${webapp.path}/WEB-INF/classes" />
      <mkdir dir="${webapp.path}/WEB-INF/lib" />

        <javac destdir="${webapp.path}/WEB-INF/classes" optimize="off" debug="on"
             failonerror="false" srcdir="${javacode.path}/jsps" excludes="**/*.smap">
            <classpath>
               <pathelement location="${webapp.path}/WEB-INF/classes" />
               <fileset dir="${webapp.path}/WEB-INF/lib">
                  <include name="*.jar" />
               </fileset>
               <pathelement location="${tomcat.home}/common/classes" />
               <fileset dir="${tomcat.home}/common/lib">
                  <include name="*.jar" />
               </fileset>
               <pathelement location="${tomcat.home}/shared/classes" />
               <fileset dir="${tomcat.home}/shared/lib">
                  <include name="*.jar" />
               </fileset>
               <fileset dir="${tomcat.home}/bin">
                  <include name="*.jar" />
               </fileset>
            </classpath>
            <include name="**" />
            <exclude name="tags/**" />
      </javac>

   </target>

   <target name="all" depends="jspc,compile">
   </target>

   <target name="cleanup">
      <delete>
         <fileset dir="${javacode.path}/jsps" />
         <fileset dir="${webapp.path}/WEB-INF/classes/org/apache/jsp" />
      </delete>
   </target>

</project>
```

3.1.3 An Example of Translating a JSP to a Servlet

Listing 2 shows an example of a JSP page that is translated to an equivalent Java Servlet presented in Listing 3. In this example, the translated code only encapsulates the HTML tags in terms of string literals that will be handled by the Web container.

Listing 2: An example JSP page

```
<HTML>
<HEAD><TITLE>Powers of 2</TITLE></HEAD>
<BODY>
<CENTER>
<H2>Behold The Powers Of 2</H2>
</CENTER>
<TABLE BORDER="2" ALIGN="center">
<TH>Exponent</TH>
<TH>2^Exponent</TH>
<% for (int i=0; i<10; i++) {%>
<TR>
<TD><%= i%></TD>
<TD><%= Math.pow(2, i)%></TD>
</TR>
<% } //end for loop %>
</TABLE>
</BODY>
</HTML>
```

Listing 3: The resulting Java Servlet based on the example JSP page

```
package myPakage;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PowersOf2 extends HttpServlet
{
public void service(HttpServletRequest request,
HttpServletResponse response)
throws IOException, ServletException
{
response.setContentType("text/html");
ServletOutputStream out = response.getOutputStream();
out.print("<HTML>");
out.print("<HEAD><TITLE>Powers of 2</TITLE></HEAD>");
out.print("<BODY>");
out.print("<CENTER>");
out.print("<H2>Behold The Powers Of 2</H2>");
out.print("</CENTER>");
out.print("<TABLE BORDER='2' ALIGN='center'>");
out.print("<TH>Exponent</TH><TH>2^Exponent</TH>");
for (int i=0; i<10; i++)
{
out.print("<TR><TD>" + i + "</TD>");
out.print("<TD>" + Math.pow(2, i) + "</TD>");
out.print("</TR>");
} //end for loop
out.print("</TABLE></BODY></HTML>");
out.close();
}
}
```

3.2 Identification of KDM Model of Java Code Using MoDisco

In the previous section, we translated JSP pages into Java code based on Servlets. In this section, we show how to identify the KDM model of these translated Servlets.

The *MoDisco* tool[6] provides a set of *APIs* and *Discoverers* that allow to extract KDM models of a given Java project based on the static analysis of its source code. The extracted model contains the complete Abstract Syntax Tree (AST) of all statements in the source code.

MoDisco offers a *Java Discoverer* as an Eclipse plug-in that allows one to easily extract the KDM model. One can follow these steps to get a KDM model.

1. **Adding Dependencies of MoDisco Plug-ins:** One has to added dependencies to three plug-ins in his/her project at the *Require-Bundle* in the *Manifest.MF* file[7]. These are:

    - *org.eclipse.gmt.modisco.java*
    - *org.eclipse.modisco.java.discoverer*
    - *org.eclipse.modisco.infra.discovery.core*

2. **Selecting the Good Discoverer Based on Your Input:** MoDisco's APIs offer a set of classes such that each one takes a different input (e.g., project, class etc.). Table 1 shows the list of MoDisco's discoverers and their required input instances. One has to decide what is the discoverer that confirms to his/her input.

    Table 1: List of MoDisco discoverers and their inputs

Dicoverer Class	**Required Input**
DiscoverJavaModelFromJavaProject	IJavaProject
DiscoverJavaModelFromProject	IProject
DiscoverJavaModelFromClassFile	IClassFile
DiscoverJavaModelFromLibrary	IPackageFragmentRoot

3. **Implementing the KDM Discoverer in Your Code:** One has to create an instance of the selected discoverer and give it the input. Listing 4 shows an example of code that implements a MoDisco discoverer from a given Java project.

Listing 4: Example of code that implements a MoDisco discoverer

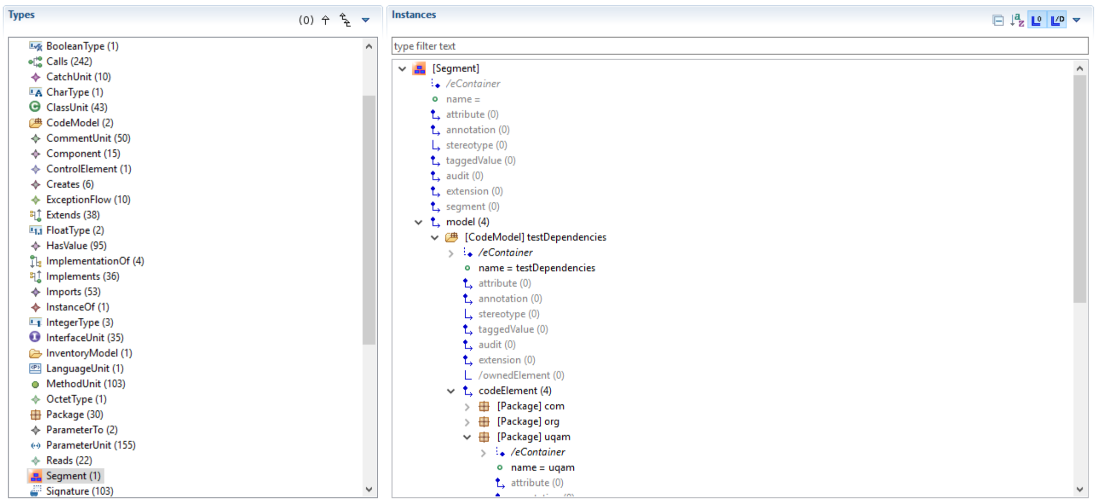
```
...
DiscoverJavaModelFromJavaProject discoverer = new DiscoverJavaModelFromJavaProject();
javaDiscoverer.discoverElement(javaProject, monitor);
Resource javaResource = javaDiscoverer.getTargetModel();
...
```

Figure 4 presents an example of a KDM model extracted based on the static analysis of the source code of a given Java project.

For more information about MoDisco, we refer to main the official website of MoDisco [4] that contains the presentation and the documentation of MoDisco, and the MoDisco's developer forum [5] that provides solutions of a bunch of problems that the users of MoDisco faced.

[6]This video shows how to install MoDisco: https://www.youtube.com/watch?v=9PAspfzJn2E

[7]See this to know how to add dependencies: https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fguide%2Ftools%2Feditors%2Fmanifest_editor%2Fdependencies.htm



```
Types                                   (0)
BooleanType (1)
Calls (242)
CatchUnit (10)
CharType (1)
ClassUnit (43)
CodeModel (2)
CommentUnit (50)
Component (15)
ControlElement (1)
Creates (6)
ExceptionFlow (10)
Extends (38)
FloatType (2)
HasValue (95)
ImplementationOf (4)
Implements (36)
Imports (53)
InstanceOf (1)
IntegerType (3)
InterfaceUnit (35)
InventoryModel (1)
LanguageUnit (1)
MethodUnit (103)
OctetType (1)
Package (30)
ParameterTo (2)
ParameterUnit (155)
Reads (22)
Segment (1)
Signature (103)
```

```
Instances
type filter text
[Segment]
    /eContainer
    name =
    attribute (0)
    annotation (0)
    stereotype (0)
    taggedValue (0)
    audit (0)
    extension (0)
    segment (0)
    model (4)
        [CodeModel] testDependencies
            /eContainer
            name = testDependencies
            attribute (0)
            annotation (0)
            stereotype (0)
            taggedValue (0)
            audit (0)
            extension (0)
            /ownedElement (0)
            codeElement (4)
                [Package] com
                [Package] org
                [Package] uqam
                    /eContainer
                    name = uqam
                    attribute (0)
```

Figure 4: Example of a KDM Model extracted from Java source code

4 The Codification of Dependencies Related to JSP Tags

There are a set of dependencies that are not recovered as a result of the previous step. Such dependencies are related to some tags that are embedded in string literals.

In this section, we present (1) the set of tags related to dependencies in JSP pages, (2) the identification of related dependencies by parsing these tags, and (3) the representation of these identified dependencies in the KDM model identified based on the previous step (c.f. Section 3).

4.1 The Set of Tags Related to Dependencies in JSP

In [6], we identified the set of JSP tags that are used to make dependencies to other server pages[8]. These are:

1. **The HTML form** (*<form action="/myPage.jsp" method="get">*) that forwards requests to server pages. The URL of the target server page URL is specified in the ***action*** attribute and the ***method*** attribute defines the request type; *get*, *put* or *post*.

2. **The *jsp:include* action tag** (*<jsp:include page="/myPage.jsp." flush="true" />*) that includes the content of server page of the relative-URL attached to the ***page*** attribute.

3. **The include directive** tag (*<%@ include file="/myPage.jsp" %>* or *<jsp:directive.include file="/myPage.jsp" />*) that merges the content of other server pages. The ***file*** attribute defines the related server page URL.

4. **The *jsp:forward* action tag** (*<jsp:forward page="myPage.jsp" />*) that forwards requests to other server pages. The target server page is determined based on a relative-URL attached to the ***page*** attribute.

[8]For more details about these tags, please refer to [6]

5. **The *page* directive tag** (*<%@ page errorPage="errorPage.jsp" %>* or *<jsp:directive.page errorPage="errorPage.jsp"/>*) that refer to a server page to be called in the case of exceptions. The ***errorPage*** attribute defines the URL of this server page.

6. **The *href* tag** (*<a href="https://www.uqam.ca">*) that makes a link to an URL.

7. **Two tags from the stander JSTL core tag library** that contains two interesting tags: the ***<c:redirect url="relative-URL"/>*** and the ***<c:url value="/myPage.jsp"*** tag.

The set of tags and their interesting attributes are shown in Table 2.

Table 2: Summarization of the list of tags and their attributes related to dependencies

Tags	**Attributes**
<form>	action, method
<jsp:include>	page
<%@ include>	file
<jsp:directive.include>	file
<jsp:forward>	page
<%@ page %>	errorPage
<jsp:directive.page>	errorPage
<a>	href
<c:redirect>	url
<c:url>	value

4.2 The Identification of Related Dependencies by Parsing Tags

We identify the dependencies of each JSP page based on: (1) the extraction of a set of URLs invoked by the JSP page, and (2) the mapping of each URL to the corresponding server page.

4.2.1 The Extraction of a Set of URLs Invoked by Each JSP Page

We develop a tool that parses JSP pages to identify a set of URLs of server pages that each JSP page depends on. Our tool relies on the *JSP Model* offered by MoDisco. As it is shown in the JSP meta-model used by MoDisco and presented in Figure 5, the JSP model only represents tags of each JSP page in terms of XML tags. However, it does not identify dependencies between JSP tags and across the JSP pages. In addition, it does not make dependencies with the other J2EE technologies including Beans components and other Java source code.

Instead of dealing with textual JSP code, we use this JSP model as intermediate representation that facilitates the parsing process of JSP pages. Then, we propose an algorithm to travel through this JSP model to identify the set of URLs invoked by each JSP page based on the list presented in Table 2.

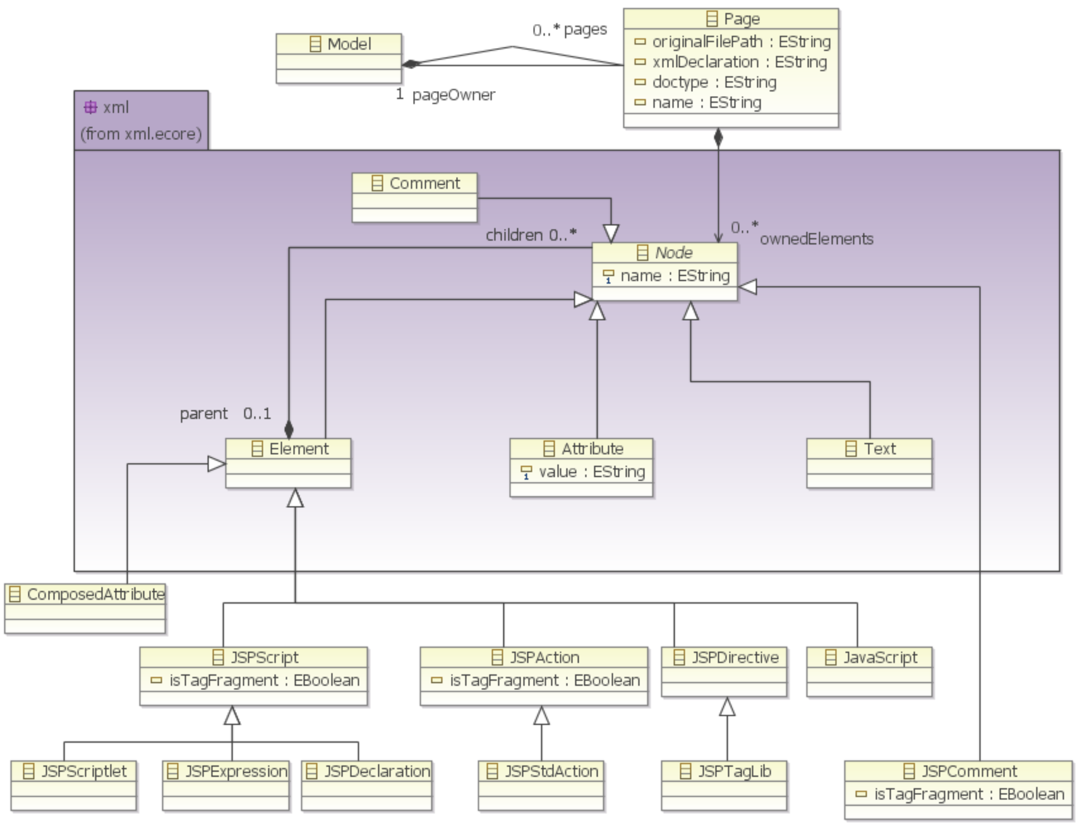


Figure 5: The metamodel of the JSP Model of MoDisco [4]

A JSP model can be programmatically identified based on the *JSP Discoverer* presented in Listing 5. An example of a JSP model extracted based on MoDisco is presented in Figure 6.

Listing 5: Example of code that implements the MoDisco's JSP discoverer

```
...
DiscoverJspModelFromJavaElement discoverer = new DiscoverJspModelFromJavaElement();
discoverer.discoverElement(javaProject, monitor);
Resource jspResource = discoverer.getTargetModel();
...
```

4.2.2 The Mapping of URLs to the Corresponding Server Pages

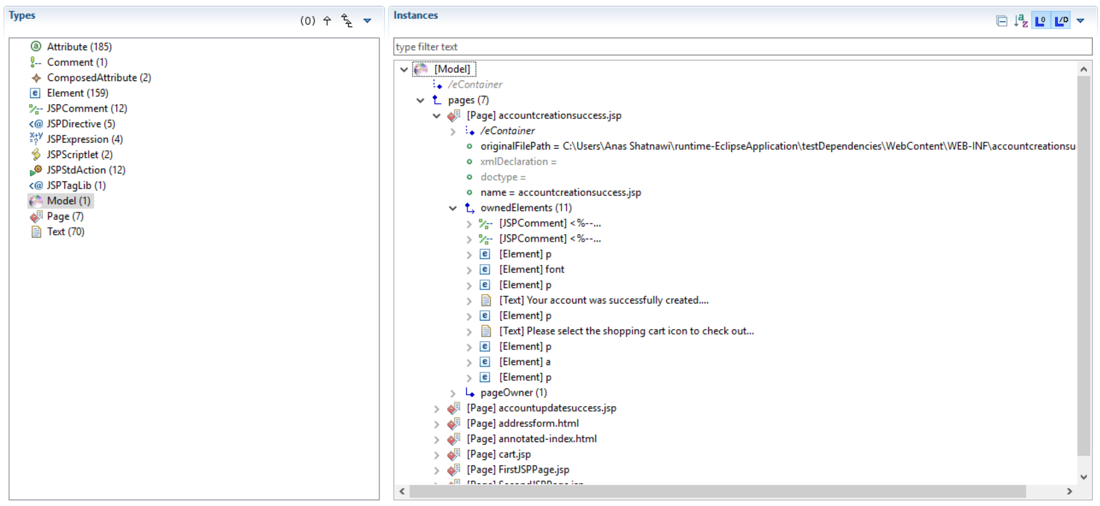
Figure 7 explains the process of mapping URLs to server pages. The URLs are mapped to server pages either using the *web.xml*[9] file (for Servlets, JSPs and JSFs) and/or *@WebServlet* annotation[10] (for Servelts) [6]. Consequently, it is essential to visit (and parse) *web.xml* and *@WebServlet* annotations.

In *web.xml*, we consider the five elements to identify mappings between server pages and relative-URLs, namely: *<Servlet>*, *<Servlet-mapping>*, *<Servlet-class>*, *<jsp-file>* and *<url-pattern>*. Then, we developed a parser[11] to extract

[9]The *web.xml* is located in the *WEB-INF/* directory of a JEE application

[10]Starting from Servlet 3.0 specification, the *@WebServlet* annotations are located within the Servlet's source code

[11]Based on the SAX2 APIs: http://www.java2s.com/Code/Java/Servlets/ParseawebxmlfileusingtheSAX2API.htm

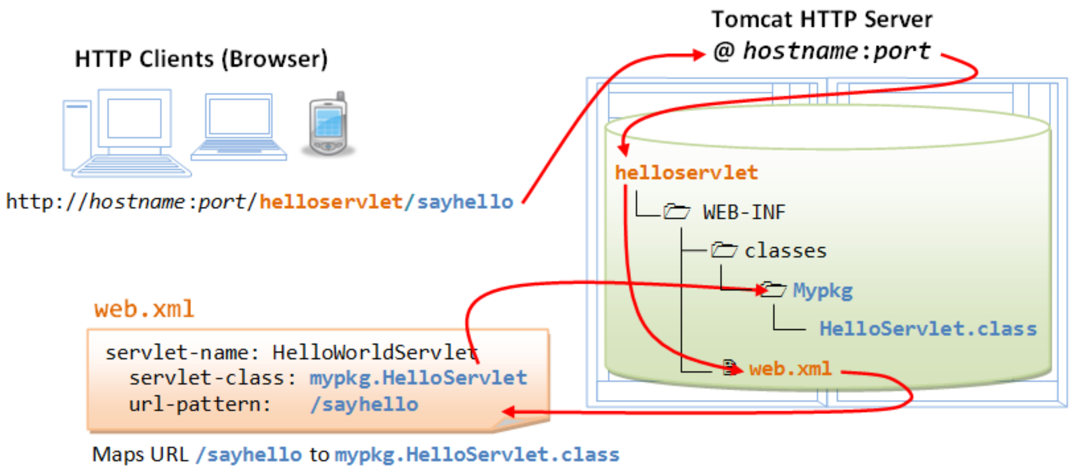


```
Types                                   (0)
Attribute (185)
Comment (1)
ComposedAttribute (2)
Element (159)
JSPComment (12)
JSPDirective (5)
JSPExpression (4)
JSPScriptlet (2)
JSPStdAction (12)
JSPTagLib (1)
Model (1)
Page (7)
Text (70)
```

```
Instances
type filter text
[Model]
    /eContainer
  pages (7)
    [Page] accountcreationsuccess.jsp
      /eContainer
      originalFilePath = C:\Users\Anas Shatnawi\runtime-EclipseApplication\testDependencies\WebContent\WEB-INF\accountcreationsu
      xmlDeclaration =
      doctype =
      name = accountcreationsuccess.jsp
      ownedElements (11)
        [JSPComment] <%--...
        [JSPComment] <%--...
        [Element] p
        [Element] font
        [Element] p
        [Text] Your account was successfully created....
        [Element] p
        [Text] Please select the shopping cart icon to check out...
        [Element] p
        [Element] a
        [Element] p
      pageOwner (1)
    [Page] accountupdatesuccess.jsp
    [Page] addressform.html
    [Page] annotated-index.html
    [Page] cart.jsp
    [Page] FirstJSPPage.jsp
```

Figure 6: Example of a JSP Model extracted based on MoDisco

the needed from the *web.xml* file. Next, we built a look-up table that maps each URL to its corresponding server page(s).



HTTP Clients (Browser)
http://hostname:port/helloservlet/sayhello
Tomcat HTTP Server
@ hostname:port
helloservlet
WEB-INF
classes
Mypkg
HelloServlet.class
web.xml
web.xml
servlet-name: HelloWorldServlet
servlet-class: mypkg.HelloServlet
url-pattern: /sayhello
Maps URL /sayhello to mypkg.HelloServlet.class

Figure 7: Mapping URLs to server pages

4.3 The Representation of Dependencies of JSP Tags in KDM Model

In the previous steps, we identified the list of server pages that each JSP depends on. Now, we want to modify the resulting KDM model from the first step (c.f., Section 3) to add dependencies of server pages. We perform as follows:

1. For each JSP page, we identify the KDM *ClassUnit* instances that is related to the current JSP page and the list of target server pages in the KDM model[12]. We rely on a sequential search algorithm to identify these *ClassUnit* instances.

[12]KDM *ClassUnit* instances were generated by the translated Servlets

2. Then, in the *service* method of the current JSP, we create new method invocations to the *service()* methods of *ClassUnit* instances of the corresponding server pages. Listing 6 shows an example of code that add a method invocation between two ClassUnit instances of two JSP pages.

Listing 6: Example of code that add method invocation between *ClassUnit* instances of two JSP pages

```
public void addMethodCallBetweenClasses(ClassUnit caller, ClassUnit targetClass) {
  if  (caller == null){
    return;
  }
  EList<CodeItem> jspServletKdm = caller.getCodeElement();
  for (CodeItem cItem : jspServletKdm) {
    if (cItem instanceof MethodUnit) {
      // select _jspService() method
      if (cItem.getName().equals("_jspService")) {
        EList<AbstractCodeElement> mElements = ((MethodUnit) cItem).getCodeElement();
        for (AbstractCodeElement element : mElements) {
          if (element instanceof BlockUnit) {
            CodeRelationship call = CodeFactory.eINSTANCE.createCodeRelationship();
            call.setTo(targetClass);
            call.setFrom(caller);
            CodeElement actionElement = CodeFactory.eINSTANCE.createCodeElement();
            actionElement.setName("newCall");
            actionElement.getCodeRelation().add((AbstractCodeRelationship) call);
            ((BlockUnit) element).getCodeElement().add(actionElement);
            element.getCodeRelation().add(call);
          }
        }
      }
    }
  }
}
```

5 Conclusion

Existing tools do not identify KDM models of JSP pages. Thus, we proposed, in this report, an approach that identifies a KDM model of JSP pages.

We first translated JSP pages to Java Servlets to reduce the problem space by mapping our problem to an already solved one. To do so, we defined *ant* code based on the *Jasper* tool provided by Apache Tomcat Server. Then, we developed a set of tools that:

- Identify the list of URLs invoked by each JSP page.
- Extract the mapping of URLs to server pages based on parsing the *web.xml* and *@WebServlet*.
- Read and modify the KDM model to include the JSP dependencies.

As a future direction, we want to develop an approach that identified KDM model for other server pages such as JSFs.

References

[1] The Apache Tomcat® software. http://tomcat.apache.org/, last access: July 1st 2017.

[2] Enterprise Edition 7 SDK Oracle: Java Platform. http://www.oracle.com/technetwork/java/javaee/downloads/java-ee-sdk-7-downloads-1956236.html, last access: July 1st 2017.

[3] Jasper: a tool to translate JSPs to Servlets. https://tomcat.apache.org/tomcat-7.0-doc/jasper-howto.html, last access: July 1st 2017.

[4] MoDisco: a reverse engineering tool. http://www.eclipse.org/modisco/, last access: July 1st 2017.

[5] The Developers Forum of MoDisco. https://www.eclipse.org/forums/index.php/f/21/, last access: July 1st 2017.

[6] Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Ghizlane El Boussaidi, Yann-Gaël Guéhéneuc, and Privat Jean Moha, Naouel. How to Implement Dependencies in Server Pages of JEE Web Applications. Technical report, LATECE Laboratory, Université du Québec á Montréal, Canada, July 2017.