© 2014 IJIRT | Volume 1 Issue 6 | ISSN : 2349-6002

JAVA SERVER PAGES

Rakesh Sondal, Sheena Batra
DCE Gurgaon, Haryana

***Abstract*— Java Server Pages or JSP for short is Sun's solution for developing dynamic web sites. JSP provides advanced and excellent server side scripting support for creating more realistic database driven web application content. JSP technology is an extension to Java servlet technology. JSP works in tandem with HTML code separating logic from the static elements , thus making HTML more functional. Before running, JSP is translated into Java servlet and then it processes HTTP requests and reply to it. JSP technology is more convenient to work with as it allows developers to directly insert java code into jsp file, making development process easy and simple. JSP technology directly support using JavaBeans components with JSP language elements. JSP pages are efficient as it loads into the web server's memory on receiving the request for the very first time only and next calls being served within short period of time. In today's environment most web sites servers dynamic pages are based on user's request. More wide use of JSP is because of its platform independent nature. This paper will enlight the unknown facts about JSP technology and how efficiently it can be brought to use in collaboration with modern systems.**

I. INTRODUCTION

Java Servlets and JSP (JavaServer Pages) constitute a widely used platform for Web application development. Applications that are developed using these or related technologies are typically structured as collections of program fragments (servlets or JSP pages) that receive user input, produce HTML or XML output, and interact with databases. These fragments are connected via forms and links in the generated pages, using deployment descriptors to declaratively map URLs to program fragments.This way of structuringapplications
causes many challenges to the programmer. In particular, it is difficult to ensure, at compile time, the following desirable properties:
– all output should be well-formed and valid XML (according to, for example, the schema for XHTML 1.0);
– the forms and fields that are produced by one program fragment that generates an XHTML page should always match what is expected by another program fragment that takes care of receiving the user input; and
– session attributes that one program fragment expects to be present should always have been set previously in the session.
Our aim is to develop a program analysis system that can automatically check these properties for a given Web application

II. JSP BASICS AND CUSTOM TAGS

A JSP page contains template text and JSP elements. All content that is not a JSP element is called template text. The template text can be any text, such as HTML, XML, WML, or even plain text, which is passed directly through to the browser. The JSP elements, which are used to generate dynamic content in the page, include directives, standard actions, custom actions, standard tag library tags, (JSTL) scripting elements and JavaBean components. Standard actions use the prefix *jsp*, such as the <jsp:useBean> and <jsp:getProperty> actions, which are use to create beans, access bean properties, and invoke other pages. But there are still many actions involved in manipulating page content not covered by standard actions, but covered by custom actions (also called custom tags) and JSTL. When developing a custom tag library, two components must be developed: the implementation of the tags in Java and an XML file called the tag library descriptor. Each individual tag is implemented as a Java class called a taghandler. A tag handler defines the tag's behavior, which must implement one of the interfaces defined in the package javax.servlet.jsp.tagext . The tag library descriptor maps each tag to the appropriate tag handler class and describes the attributes supported by each tag. Tags are made available within a JSP page via the taglib directive.The tag libbrary *xxxlib* has a local name within the JSP file of *mylib* and the custom tag is named *cdtitle*. The implementation of the tag is in the file *CdtitleTag.class*. The JSP container uses the TLD to find the information it needs to generate code for invoking the correct tag handler class when it encounters an action element with a matching prefix. Sun provides a standard tag library which implements statement level tags for control flow and database operations. Our approach produces higher level tags abstracting the business logic of the web application. As well, JSP 2.0 provides an alternate interface for custom tags, the simpleTag interface. It simplifies the implementation, and tags need only implement a single method. Our technique works with both interfaces.

III. GENERAL APPROACH

In practice, HTML code and Java code are tightly interweaved in existing JSP web applications. Java code is embedded in HTML code and HTML tags are embedded in Java code. Before we discuss the code transformation, we must first examine the elements that are affected by the transform.
JSP scripting elements. Scripting elements allow us to add pieces of Java code in a JSP page, which have three types: *scriptlets*, *expressions*, and *formal declarations*. The scripting elements in a JSP page will be executed each time the page is requested. It is better to move and encapsulate Java code implementing business logic into

appropriate custom tags.

HTML content embedded in *out.print()*. The Java code sequence *out.print(…);* is a commonly-used simple method to generate HTML content from within a scriptlet. The use of this code has two disadvantages. One is that it needs extra effort to create and maintain

HTML pages for application programmers. The other is that content authors (or web page designers) must understand the embedded code or must ask the application programmers to make any changes to the scriptlet. Before we perform the transformation, we extract the HTML contentfrom the *out.print()* and *out.println()* statements and put the content directly into the page. This changes the code to an HTML segment that is nested within the scriplet. It starts with a JSP end tag (%>) and ends with a JSP start tag (<%), allowing HTML segments to appear inside interesting JSP elements and is used to handle "the lake inside an island" parse. As a result, the extracted HTML code can be included directly into the web pages without *out.print()* calls and page designers can make changes to the HTML contents without risk of breaking the Java code.

JavaBean action elements. A JavaBean component is a Java class that has a no-argument constructor and conforms to the JavaBean coding conventions. There are three kinds of JSP standard action elements which allow developers to use JavaBean components :

<*jsp:useBean*> instantiates a Java Bean and makes it available in a page; <*jsp:getProperty*> gets a property value from a JavaBean and adds it to the response; <*jsp:setProperty*> sets all properties value in a JavaBean with names matching the names of the parameters received from the request. These three action elements along with other scriptletsare all functions which are more suitable for an application programmer. The web page designer need not knowhow to instantiate a Java Bean or how to set properties of the bean. The page designer only needs to know the properties of a Java bean component in applications where a separate servlet instantiates a bean and passes it to a JSP page for display. To this end, we change the three action elements into valid Java code enclosed within scriptlet

tags (<*%…%*>) before the general transformation.

JSP page directives. JSP page directives are usually found at the top of a JSP page and always enclosed within directive tags (<*%@ … %*>). There can be any number of page directives within a JSP page, but the attribute/value pair must be unique. The cases that affects the transform is the case where the page imports a Java package. For

example, <*%@ page import ="java.sql.ResultSet" %*> This directive imports the class ResultSet from the package *java.sql* into the page. As this class should also be imported into our newly created custom tags, we change the page directive into a *scriptlet*. For example, the directive above becomes <*% import java.sql.ResultSet;*

%>. We can then transform this valid import statement as part of the general transform.

IV. IMPLEMENTATION

In this section, we briefly describe the implementation of the transformation. Most of the process is implemented using the TXL language, which is a pure functional programming language particularly designed tto support rule-based source-to-source transformation. Transformation process, which includes five

phases. The preprocessing phase normalizes the source

code. It also performs some comment and lexical preprocessing.

The grouping phase performs an analysis and annotates each line of normalized source code with a tag id identifying the custom tag to which the source code will belong. The first part of the analysis identifies the cases that we have identified in this paper. It identifies control statements that contain HTML/JavaScript and the first

statement of Java sequences within the template text that must be given their own tags. Thus, the basic structure of

the tags is identified. The second phase of grouping assigns

the remaining statements to one of the identified tags. The tag id generated in the group phase is mapped to a reasonable user name such as *invalidLogin* or *CDTitle* by a web interface in the tag naming phase. The code transformation phase uses the markup from the group phase and the mapping from the tag naming phase to generate the three outputs of the process. These are the modernized JSP pages, the tag library description file, and the custom tag classes. The final post-processing phase deals with final touchups such as fixing comments. The whole process is automated, except the tag naming

phase where the human assistance is required. The details of the implementation, particularly, the markup approach and the transformations are described elsewhere. We have tested our system on 3 small systems to date consisting of an online music store, a mini weblog application and a guest book application. Two were obtained

from within Queen's, the other is a sample systemdownloaded from the internet. The systems comprise a total of 14 JSP pages containing a total of 682 lines of mixed JSP and HTML. The resulting pages contain 362 lines of tags and HTML. 74 custom tag classes were generated. Currently each JSP expression is translated into its own

custom tag. A simple optimization is to fold the simple JSP expressions into one simple tag. This would eliminate 23 custom tag classes.

```
<table border>
<%
String title;
String price;
ResultSet (loopvar.next() )
{
%>
<tr>
<%
title=loopvar.getString(1);
```

```
%>
<td> Title</td>
<td><%=title%></td>
</tr>
<tr>
<%
price=loopvar.getString(2);
%>
<td>Price</td>
<td> <%=currency.format(price)%> </td>
</tr>
<%
}
%>
```

V. CONTRIBUTION

Our contributions are the following:

– We show how to obtain a context-free grammar that conservatively approximates the possible output of servlet/JSP applications using a variant of the Java string analysis

– On top of the string analysis, we apply theory of balanced grammars by Knuth and grammar approximations by Mohri and Nederhof to check that the output is always well-formed XML.

– On top of the well-formedness checking, we show how a balanced contextfree grammar can be converted into an XML graph, which is subsequently validated relative to an XML schema using an existing algorithm.

– By analyzing the form and link elements that appear in the XML graph together with the deployment descriptor of the application, we explain how to obtain an inter-servlet control flow graph of the application.

– Based on the knowledge of the control flow, we give examples of derived analyses for checking that form fields and session state are used consistently.

Together, the above components form a coherent analysis system for reasoning about the behavior of Web application that are built using Java Servlets and JSP. The system has a *front-end* that converts from Java code to context-free grammars and a *back-end* that converts context-free grammars to XML graphs and checks well-formedness, validity, and other correctness properties. Our approach can be viewed as combining and extending techniques from the JWIG and Xact projects and applying them to a mainstream Web application development framework. Perhaps surprisingly, the analysis of well-formedness and validity can be made both sound and complete relative to the grammar being produced in the frontend. (The completeness, however, relies on an assumption that certain welldefined contrived situations do not occur in the program being analyzed). The goal of the present paper is to outline our analysis system, with particular focus on the construction of context-free grammars and the translation from context-free grammars to XML graphs. We base our presentation on a running example. The system is at the time of writing not yet fully implemented. Although we here focus on Java-basedWeb applications, we are not relying on language features that are specific to Java. In particular, the approach we present could also be applied to the .NET or PHP platforms where Web applications are typically also built from loosely connected program fragments that each produce XHTML output and receive form input

CONCLUSION

The implementation of our transformation is a greedy approach. It attempts to group as many statements as possible into each tag. Each web page is also processed independently. One potentially extension is to identify clones between pages, separating them in to separate tags. One example is session management code common to multiple pages. In this paper, we have presented a set of transforms that can be used to implement the separation of the presentation and business logic for existing JSP-based web applications. The transforms restructure the web applications by moving Java code embedded in JSP pages into custom tags without changing the original functionalities and user interfaces of the applications. The interesting information required for this restructuring is contained not only in the multiple languages themselves but also in the way they are coupled.

An advantage of our Java code transformation is that all business logic intensive Java code in JSP pages is moved and encapsulated into custom tags and all elements for presentation are kept in pages, which helps to reduce the complexity of web applications and helps make the restructured applications more reusable and maintainable.

REFRENCES

[1] Hans Bergsten, *JavaServer Pages*, O'Reilly 2002.

[2] T. Bodhuin, E. Guardabascio, M.Tortorella, "Migrating COBOL Systems to the WEB by Using the MVC Design Pattern", Proc Working conference on Reverse Engineering, Richmond, Virginia, pp 329-338.

[3] C. Boldyreff, R. Kewish, "ReverseEngineering to Achieve Maintainable WWW Sites", Working Conference on Reverse Engineering, Stuttgart, Germany, Oct. 2001

[4] J. Cordy, "TXL – A Language for Programming Language Tools and Applications", Proc ACM International Workshop on Language Descriptions, Tools and Applications, Edinburg, Scotland, January 2005, pp. 3-31.

[5] A. van Deursen, T. Kuipers, "Building Documentation Generators", Proc International Conference on Software Maintenance, Oxford, England, 1999, pp 40-49.
[6] D. Draheim, E. Fehr, G. Weber, "JSPick – A Server Pages Design Recovery Tool", Proc 7th European Conference on Software Maintenance and Reengineering, Benevento, Italy, March 2003, pp 230-238.
[7] A. Hassan, R. Holt, "Architecture Recovery of Web Applictions",
Proc International Conference on Software Engineering, Orlando, Florida, May 2002, p 19-25.
[8] A. Hassan, R. Holt, "Migrating Web Frameworks Using Water Transformations", Proc International Computer Software and Application Conference, Dallas, Nov. 2003.