Data flow analysis and testing of JSP-based Web applications

Chien-Hung Liu *

Department of Computer Science and Information Engineering, National Taipei University of Technology, 1, Sec. 3, Chung-Hsiao E. Rd., Taipei 106, Taiwan

Received 14 June 2006; accepted 14 June 2006
Available online 14 August 2006

Abstract

Web applications often rely on server-side scripts to handle HTTP requests, to generate dynamic contents, and to interact with other components. The server-side scripts are usually not checked by any compiler and, hence, can be error-prone. In this paper, we adapt traditional data flow testing techniques into the context of Java Server Pages (JSP), a very popular server-side script for developing Web applications with Java Technology. We point out that the JSP implicit objects and action tags can introduce several unique data flow test artifacts which need to be addressed. A test model is presented to capture the data flow information of JSP pages with considerations of various implicit objects and action tags. Based on the test model, we describe an approach to compute the intraprocedural, interprocedural, and sessional data flow test paths for uncovering the data anomalies of JSP pages.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Web application testing; Data flow testing; Software testing; JSP testing

1. Introduction

With the wide acceptance of Internet, Web applications have become popular vehicles for conducting transactions, delivering services, and acquiring information. As a result, there is a great demand on sophisticated and high-quality Web applications. However, Web applications are often developed under a short time-to-market pressure without following traditional software engineering discipline [14]. The lacks of rigor, systematic approach, and quality assurance control have raised a concern about the quality and reliability of Web applications in spite of the testing challenges introduced by Web technologies [5].

Recently, several approaches have been proposed to address Web application testing [1,2,4,7–12,16,18,21,22]. Most of the approaches focus on recovering the architectures of Web applications at page level so as to provide testers a roadmap to exercise the Web pages and their relationships. However, in order to provide sufficient code coverage, traditional structural testing is still required for Web applications. Thus, in this paper, we aim to make an effort on data flow analyzing and testing of Web applications that are developed using JSP pages, a very popular server-side script language for developing Web applications with Java technology.

JSP pages have been widely used in Java-based Web applications to handle HTTP requests, to interact with Java components like Java beans, and to generate dynamic pages. It is important to ensure that the JSP pages are written correctly and their interactions with other components are handled properly. However, JSP pages usually mix up scripts (i.e., JSP scriptlets) with HTML statements in order to generate dynamic pages. This makes JSP pages difficult to understand and test.

Moreover, as a scripting language, JSP pages do not have any compiler checking and, hence, can be error-prone. Although there have existed several test harnesses for testing JSP pages, such as HTTPUnit and Cactus [17], JSP page testing is considered difficult and test cases are still designed in ad hoc process. Most important, JSP pages have introduced a set of XML-like action tags and implicit objects [3]. These action tags and implicit objects can raise
some concerns when the internal program logic of JSP pages is exercised using traditional data flow testing techniques [6,15].

In this paper, we identify and analyze the possible data flow test artifacts introduced by the JSP pages. A test model is proposed to abstract the data flow information of various JSP implicit objects and action tags. An approach for computing the data flow test paths involving the implicit objects and action tags is described and illustrated.

The paper is organized as follows. Section 2 briefly reviews existing Web application testing approaches. Section 3 describes the data flow test artifacts raised by JSP implicit objects and action tags. Section 4 proposes a data flow test model to represent these JSP test artifacts. Section 5 illustrates an approach to compute the intraprocedural, interprocedural, and sessional data flow information of JSP pages. The last section summarizes the conclusions.

2. Related work

Recently, a few more Web application testing approaches have been proposed. Yang et al. [22] expand traditional software testing architecture to support Web application testing. A set of tools is developed to help analyze documents, develop test cases, execute tests, monitor failures, and support test measurement. Kung et al. [8,10] present a test model that abstracts the unstructured Web applications in terms of objects and relationships. They consider the HTML documents as objects and analyze their possible data flow interactions with other components of the Web applications. A data flow test strategy is presented to select test paths for exercising scripts from intra-object, inter-object, and inter-client perspectives.

Ricca and Tonella [16] describe an analysis model that captures the Web pages, forms, frames, and their relationships of Web applications. The model extracts both static and dynamic Web pages of a Web application. From the model, test cases can be derived to test the data flow among the Web pages. Lucca et al. [11] propose a complex test model to represent various entities of a Web application. Based on their model, a strategy is suggested to generate preliminary test cases for unit and integration testing of Web applications.

Wu and Offutt [21] describe a composition technique and several rules for structuring static and dynamic pages using the atomic contents of a page. Based on the user interactions, they derive test cases as sequences of static pages and possible compositions of the atomic elements. Offutt et al. [12] present an approach to test Web applications by skipping the client-side input validation. They bypass the input validation from the input value, input parameter, and control flow perspectives to test the vulnerability of Web application. The empirical results are presented to demonstrate the usefulness of the proposed approach.

Andrews et al. [1] model the state dependent behavior of Web applications using hierarchical FSMs. Basically, a Web application is partitioned into clusters. Each cluster is represented in terms of a FSM, where states represent Web pages or HTML forms and transitions represent user inputs and actions. Test sequences are then generated from the aggregated FSMs. In addition, the proposed method employs input constraints to reduce the state space explosion.

Benedikt et al. [2] present a tool to automate the testing process of Web applications. Basically, the tool can generate test inputs for a Web site based on a given profile. With the test inputs, the tool can automatically explore the static and dynamic pages of the Web site.

In addition to the approaches for testing the functionality of Web applications, Kallepalli and Tian [7] and Tonella and Ricca [18] exploit the usage information and failure logs gathered during the executions of Web applications to support reliability analysis. At the same time, Elbaum et al. [4] leverage the captured usage data of Web applications to produce test cases at less expense than that of the test cases generated using white-box testing techniques.

It should be pointed out that the data flow test artifacts considered in this paper is different from that in [10]. The main focus of this paper is the data flow introduced by implicit objects and action tags in the server-side JSP pages while in [10] the major concern is the data flow introduced by HTML documents among client-side scripts and the data flow caused by HTTP protocol between client and server pages.

3. Data flow test artifacts of JSP pages

Data flow testing mainly focuses on the definitions of variables and their potential uses for exploring the data anomalies of programs. The use of a variable can be classified as c-use (computation use) or p-use (predicate use). A c-use occurs whenever a variable is used in a computation or output statement while a p-use occurs whenever a variable is used in a predicate statement. Test paths of a program are selected based on the def-use chains (or definition-use chains) of variables, where a def-use chain of a variable is a path from the definition to the use of the variable without any intervening redefinition [15].

To explore the data anomalies of JSP pages, we need to consider not only the variables of the JSP pages, but also the implicit objects and action tags introduced by the JSP technology. The implicit objects, such as request and response objects, allow JSP developers to access JSP-provided services and resources without explicitly declaring the objects. In particular, some of the implicit objects have the functionality to control the input/output and the flow of JSP pages. For instance, the request object can be used to access the data supplied in client requests. The same request object can be available to all JSP pages that service a given HTTP request, which can cause data interactions among the JSP pages.

Similarly, the implicit session object is like a container that can store name-value pairs. Through the session...
object, the values of the name-value pairs can be used or changed by different JSP pages. This enables data to be accessed across JSP pages within a user session and, hence, can introduce data flow artifacts among JSP pages.

Moreover, the sendRedirect() method of the response object allows an HTTP request to be redirected to another JSP page. The redirection not only can affect the flow of data in the JSP pages, but also can introduce data interactions between two JSP pages or between a JSP page and a Java servlet. As shown in Fig. 1, the data of variables var1 and var2 can be passed from page1.jsp to page2.jsp through the response.sendRedirect() method, which introduces data interactions between page1.jsp and page2.jsp.

In addition to the implicit objects, the JSP technology introduces a standard set of XML-like action tags, such as <jsp:useBean> and <jsp:forward>. The action tags are associated with code handlers that allow JSP pages to interact with other JSP pages and Java objects. This can introduce data interactions among JSP pages and Java objects. For example, consider the login.jsp page in Fig. 2. In Line 5, the login.jsp page has a <jsp:useBean> tag that assigns a tag attribute test to a Java object TestBean. This test attribute is then used in Line 8 as an object reference to the TestBean object, where the data of variables login and passwd are passed from the login.jsp page to the test.verify() method. As a result, the <jsp:useBean> tag needs to be considered when computing the data flow information for JSP pages in order to capture the data interactions between the JSP pages and Java objects.

Moreover, in Line 12 of Fig. 2, the <jsp:forward> tag allows an HTTP request to be forwarded to the debug.jsp page with a parameter my_num. The <jsp:forward> tag not only changes the flow of the login.jsp page, but also causes data interactions between the login.jsp and debug.jsp pages. Notice that, unlike traditional function calls, the <jsp:forward> tag simply transfers the control of execution from the login.jsp page to the debug.jsp page without any return. Such unidirectional data flow must be considered so that the data flow information of JSP pages can be computed correctly.

Table 1 summarizes several essential data flow test artifacts introduced by the JSP implicit objects and action tags. It should be pointed out that, in addition to the <jsp:forward> tag, the <jsp:include> tag allows data parameters to be passed to another JSP page or a Java servlet and, hence, can introduce data interactions between JSP pages or servlets [3]. Nevertheless, the <jsp:include> tag only includes the outputs from other JSP pages or servlets into the current JSP page. The flow of control is similar to that of traditional function calls and will return back to the current JSP page.

4. Data flow test model for JSP pages

From the viewpoint of control structure, JSP pages are more like traditional procedures rather than object classes although a JSP page can contain several script functions. The reason is that a JSP page has only one entry point and all the functions defined in the JSP page can only be accessed within the JSP page. Thus, for abstracting the data flow information of JSP pages, we consider JSP pages as traditional procedures and capture their intraprocedural and interprocedural data flow test artifacts.

However, unlike traditional programs, JSP pages do not have an explicit calling-hierarchy and the invoking sequences of JSP pages can be dependent of navigation

---

**Fig. 1.** An example of data interactions via response.sendRedirect().

```java
// part of page1.jsp
1 String var1 = "my_value1"
2 String var2 = "my_value2"
3 response.sendRedirect("page2.jsp?param1=" + var1 + "+param2=" + var2);

// part of page2.jsp
4 String my_var1 = request.getParameter("param1");
5 String my_var2 = request.getParameter("param2");
```

---

**Fig. 2.** An example of data interactions caused by the JSP action tags.
paths within a user session. In particular, through accessing the implicit session object, different navigation scenarios can cause different data interactions across the JSP pages. Such data interactions can occur according to user navigation paths rather than occur based on the control flow of the JSP pages. To capture such data flow artifacts explicitly, we also model the data flow introduced by the session object.

4.1. Intraprocedural data flow model

To represent the intraprocedural data flow information for JSP pages, a control flow graph (CFG) annotated with def/use information is employed. The def-use annotated CFG is a directed graph, where nodes represent the statement blocks of JSP scripts and edges represent the execution flow between the statement blocks. In particular, the CFG is annotated what happen to the variables, implicit objects, and action tags of interest.

Note that, as described in Section 3, the response.sendRedirect(), pageContext.forward(), and <jsp:forward> tags can pose a unidirectional data flow. To capture this artifact, an “end” node is created in the CFG to represent the termination of the control flow. Thus, the intraprocedural test model may not be a single-entry-single-exit CFG. Moreover, the HTML statements in the JSP pages are considered as the output statements without any definitions or uses of the variables. In addition, each action tag block, including a start tag, zero or more tag attributes, a tag body, and an end tag, is represented as a single block of the CFG.

Fig. 3 shows the def/use annotated CFG for the login.jsp page in Fig. 2. The definitions and the uses of variables, implicit objects, and action tags are annotated at corresponding nodes. Notice that, in Line 12 of Fig. 2, all the statements starting from the <jsp:forward> tag and ending at the </jsp:forward> tag are considered a single XML block of the <jsp:forward> tag. This block is represented by node 12 of the CFG. In addition, an “end” node is created in the CFG due to the <jsp:forward> tag. At this node, the flow of control is transferred from the login.jsp page to the debug.jsp page without return, and an interprocedural data flow can be introduced.

4.2. Interprocedural data flow model

To represent the data flow information involving more than one function or one JSP page, an interprocedural control flow graph (ICFG) annotated with def/use information is employed. The ICFG has been used to model the interprocedural data flow of interacting functions in traditional programs [6,13]. To capture the interprocedural data flow among JSP pages and Java components, the construction of ICFG has to take into consideration the implicit objects and action tags in addition to function calls. The following list several situations where the interprocedural data flow can happen and the ICFGs need to be constructed for corresponding JSP pages and Java components.

<table>
<thead>
<tr>
<th>JSP implicit objects/action tags</th>
<th>Data flow test artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>can present data interactions between JSP pages that access the same request object in a given HTTP request</td>
</tr>
<tr>
<td>response.sendRedirect()</td>
<td>can change the flow of data and introduce data interactions between JSP pages or servlets</td>
</tr>
<tr>
<td><a href="">jsp:useBean</a></td>
<td>can introduce data interactions between a JSP page and a Java component</td>
</tr>
<tr>
<td><a href="">jsp:forward</a>, pageContext.forward()</td>
<td>can introduce data interactions between JSP pages and Java components</td>
</tr>
<tr>
<td><a href="">jsp:include</a></td>
<td>can introduce data interactions between JSP pages or servlets when the tag is used with the <a href="">jsp:param</a> tag</td>
</tr>
<tr>
<td><a href="">jsp:setProperty</a>, <a href="">jsp:getProperty</a></td>
<td>can introduce data interactions between a JSP page and a Java object when the tag is used with the <a href="">jsp:param</a> tag</td>
</tr>
<tr>
<td>%=</td>
<td>variables declared within the declaration tag % can be globally accessible to the JSP scriptlets and expressions</td>
</tr>
<tr>
<td>session</td>
<td>can introduce data interactions between JSP pages using session object to maintain session information</td>
</tr>
</tbody>
</table>
The “main” script or a script function of a JSP page invokes another script function within the same JSP page.

The JSP page invokes corresponding Java components through the `<jsp:getProperty>` or `<jsp:setProperty>` tag or through the direct reference to a method of a Java component.

The JSP page invokes another JSP page or Java servlet through the `<jsp:forward>` tag, the `response.sendRedirect()` method, or the `pageContext.forward()` method.

Notice that the action tags `<jsp:getProperty>`, `<jsp:setProperty>`, and `<jsp:include>` have control flow similar to that of traditional function calls. As a result, the ICFG can be constructed by connecting the nodes representing these action tags in the calling JSP pages to the entry nodes of the called Java methods or JSP pages. However, the `<jsp:forward>` tag, the `pageContext.forward()` method, and the `sendRedirect()` method cause unidirectional control flow. Thus, the construction of the ICFG needs to connect the “end” nodes corresponding to the `<jsp:forward>` tag, the `pageContext.forward()` method, and the `sendRedirect()` method in the forwarding (or redirecting) JSP pages to the entry nodes of the forwarded (or redirected) JSP pages or Java methods.

For illustration, consider the interacting JSP pages shown in Fig. 4. In Line 7, the `icfg.jsp` page can pass a request object, composed of attributes `user_x` and `my_y`, to the `fw.jsp` page with the use of the `<jsp:forward>` tag. Thus, the attributes `user_x` and `my_y` can be used and defined, respectively, in the `icfg.jsp` page (in Lines 4 and 7) and used in the `fw.jsp` page (in Lines 12 and 13). Moreover, in Line 6, the variable `x`, an alias of the attribute `user_x`, is passed to a local function `inc()` in the `icfg.jsp` page where `x` is used in Line 2. The attribute `user_x` is also passed to the `in.jsp` page in Line 8 via the same request object using the `<jsp:include>` tag and is used in Line 17.

Fig. 5 presents the def/use annotated ICFG for the interacting JSP pages shown in Fig. 4, where the shaded nodes represent the main entry and two exit nodes of the ICFG. The definitions and the uses of variables, implicit objects, and action tags are annotated at corresponding nodes. Note that, in Fig. 5, the “end” node of the `icfg.jsp` page is connected to the entry node of the `fw.jsp` page to represent the interprocedural data flow caused by the `<jsp:forward>` tag. Moreover, the control flow of the `<jsp:include>` tag is similar to that of the `inc()` function in which the flow of control will be back to the `icfg.jsp` page when the executions of the `in.jsp` page and the `inc()` function are completed.

4.3. The sessional data flow model

The JSP session implicit object can be considered as a global container that stores the name-value pairs shared by different JSP pages within a user session. The name-value pairs stored in the `session` object can be defined and used through the `session.setAttribute()` and `session.getAttribute()` methods, respectively. Since the accessing sequences of the `session` object can depend on the navigation paths,
there can exist no explicit inclusion, forward or redirection relationship among the JSP pages that access the session object. Thus, the data flow of the session object cannot be depicted simply by using the ICFG described in Section 4.2.

To represent the data flow information involving the session object, a session control flow graph (SCFG) annotated with def/use information is employed. The SCFG is constructed by connecting together the CFGs (or ICFGs) of all the JSP page clusters that involve the session object, where a JSP page cluster is a set of JSP pages interacting through function calls, \texttt{<jsp:include>} and \texttt{<jsp:forward>} tags, or \texttt{pageContext.forward()} and \texttt{response.sendRedirect()} methods. According to user’s interactions, different JSP page clusters can be invoked according to the navigation paths.

For illustration, consider the JSP pages in Fig. 6 that have data interactions caused by the session object. In Lines 6 and 7, the \texttt{ex1.jsp} page defines the attributes \texttt{userName} and \texttt{userBalance} of the session object using the \texttt{session.setAttribute()} method. These two attributes are referenced, respectively, in Lines 13 and 15 of the \texttt{ex2.jsp} page and in Lines 26 and 27 of the \texttt{ex3.jsp} page with the \texttt{session.getAttribute()} method. The value of \texttt{userBalance} is re-defined in Line 20 of the \texttt{ex2.jsp} and in Line 30 of the \texttt{ex3.jsp}. The data interactions among these pages are introduced only through the session object.

Fig. 7 shows an example of the def/use annotated SCFG for the JSP pages in Fig. 6. The SCFG is constructed by connecting together the CFGs of the page clusters corresponding to the \texttt{ex1.jsp}, \texttt{ex2.jsp}, and \texttt{ex3.jsp} pages. Since each of the JSP pages has no interprocedural data flow caused by function calls or action tags, the page cluster for each JSP page only contains the page itself.

Moreover, to model the data flow among the JSP page clusters, two additional nodes C and L are created. In particular, the node C is connected to the entry nodes of the page clusters while the exit nodes of the page clusters are connected to the node L. The node L is then connected to node C. This allows the SCFG to represent the execution paths for any permissible invocations of the page clusters.

To derive data flow information introduced by the session object, the SCFG is annotated with the \texttt{userName} and \texttt{userBalance} attributes and their aliases that can be defined and/or used in different JSP pages. Notice that the pages can be invoked in any permissible sequences according to user’s navigations.

5. The computation of definition-use chains for JSP pages

In this section, we describe how to derive the definition-use chains introduced by JSP implicit objects and action tags. Based on the test models presented in Section 4, the definition-use chains can be obtained from the intraprocedural, interprocedural, and sessional perspectives.
5.1. The definitions and uses of implicit objects and action tags

In order to explore the data anomalies involving the implicit objects and action tags, we define the “definitions” and “uses” for the attributes or Java objects associated with the implicit objects and action tags. Table 2 lists the situations where the attributes or Java objects associated with implicit objects or action tags are considered to be defined and used.

It should be noted that the attributes of the request object are usually defined in the client-side HTML documents. Therefore, to compute the def-use chains for the attributes associated with the request object, the request object is treated as a global object in a JSP page and the associated attributes are considered to be defined at the entry node of the CFG as shown in Fig. 3. However, the attribute my_num of the request object is considered to be defined at node 12 because my_num is specified by the <jsp:param> tag embedded in a <jsp:forward> tag.

Moreover, due to the <jsp:useBean> tag, the Java object test is considered to be defined at node 5. It is then used at node 8 by directly referencing the method verify() and is used at node 10 through the tag <jsp:getProperty>. Thus, from Fig. 3, we can obtain a def-use chain <entry, 6> for the attribute user_login, a def-use chain <entry, 7> for the attribute user_passwd, and def-use chains <5, 8> and <5, 10> for the test object.

The def-use chains for the local script variables login, passwd, and num can be computed from the CFG and are given in Table 3. Note that a JSP page may consist of statements involving only implicit objects, action tags, and HTML codes. This means that no local script variables are declared and used. In such a case, the data flow information of the JSP page can be computed from the implicit objects and/or action tags.

5.2. The computation of intraprocedural definition-use chains

With the definitions in Section 5.1, we can compute the intraprocedural data flow information involving the implicit objects and action tags for JSP pages. For example, consider the request object in the login.jsp page in Fig. 2. Since the attributes for the request object are defined in client-side, the attributes user_login and user_passwd of the request object are considered to be defined at the “entry” node of the CFG as shown in Fig. 3. However, the attribute my_num of the request object is considered to be defined at node 12 because my_num is specified by the <jsp:param> tag embedded in a <jsp:forward> tag.

Moreover, due to the <jsp:useBean> tag, the Java object test is considered to be defined at node 5. It is then used at node 8 by directly referencing the method verify() and is used at node 10 through the tag <jsp:getProperty>. Thus, from Fig. 3, we can obtain a def-use chain <entry, 6> for the attribute user_login, a def-use chain <entry, 7> for the attribute user_passwd, and def-use chains <5, 8> and <5, 10> for the test object.

The def-use chains for the local script variables login, passwd, and num can be computed from the CFG and are given in Table 3. Note that a JSP page may consist of statements involving only implicit objects, action tags, and HTML codes. This means that no local script variables are declared and used. In such a case, the data flow information of the JSP page can be computed from the implicit objects and/or action tags.

5.3. The computation of interprocedural definition-use chains

From the interprocedural perspective, the def-use chains for script variables and attributes associated with implicit objects and action tags that are defined in one script function and used in another script function (or JSP page) can...
be obtained from the def/use annotated ICFG. For instance, in Fig. 4, the attribute user_x of the request object can be defined in the icfg.jsp page and used in the fw.jsp and in.jsp pages. From Fig. 5, we can compute two inter-procedural def-use chains <entry, 12> and <entry, 17> for the attribute user_x. These def-use chains cover the paths from the entry node of the icfg.jsp page to the node 12 in the fw.jsp page and to node 17 in the in.jsp page.
Table 3
The intraprocedural def-use chains of the login.jsp page

<table>
<thead>
<tr>
<th>Variables, implicit objects, or action tags</th>
<th>Intraprocedural def-use chains</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>&lt;6, 8&gt;</td>
</tr>
<tr>
<td>passwd</td>
<td>&lt;7, 8&gt;</td>
</tr>
<tr>
<td>num</td>
<td>&lt;8, 12&gt;, &lt;8, (9, 10), &lt;8, 12&gt;</td>
</tr>
<tr>
<td>request.user_login</td>
<td>&lt;entry, 6&gt;</td>
</tr>
<tr>
<td>request.user_passwd</td>
<td>&lt;entry, 7&gt;</td>
</tr>
<tr>
<td>test</td>
<td>&lt;5, 8&gt;, &lt;5, 10&gt;</td>
</tr>
</tbody>
</table>

Table 4
The interprocedural def-use chains between the icfg.jsp, fw.jsp, and in.jsp pages

<table>
<thead>
<tr>
<th>Variables or implicit objects</th>
<th>Interprocedural def-use chains</th>
</tr>
</thead>
<tbody>
<tr>
<td>request.user_x</td>
<td>&lt;entry, 12&gt;, &lt;entry, 17&gt;</td>
</tr>
<tr>
<td>request.user_y</td>
<td>&lt;7, 13&gt;</td>
</tr>
<tr>
<td>x</td>
<td>&lt;4, 2&gt;</td>
</tr>
<tr>
<td>y</td>
<td>&lt;2, 7&gt;</td>
</tr>
</tbody>
</table>

In addition, the attribute _my_y_ of the request object is defined in the icfg.jsp page through the <jsp:param> tag and is used in the fw.jsp page. Thus, for the attribute _my_y_, we can obtain an interprocedural def-use chain <7, 13>. Moreover, there exist intraprocedural def-use chains for the local variables x and y that are defined and used in different script functions of the same icfg.jsp page. Table 4 shows the interprocedural def-use chains between the icfg.jsp, fw.jsp, and in.jsp pages.

5.4. The computation of session definition-use chains

From the session perspective, the def-use chains for attributes associated with the session objects can be obtained from the def/use annotated SCFG. For example, in Fig. 6, the attributes _session_ can be defined in the ex1.jsp page and used in the ex2.jsp and ex3.jsp pages. According to user navigation, this definition can be referenced in Line 15 of the ex2.jsp page and in Line 27 of the ex3.jsp page, respectively. The associated data flow paths can be obtained from the SCFG in Fig. 7 and covered by the def-use chains <7, 15> and <7, 27>, respectively.

Moreover, in Fig. 6, the value of _session_ can be redefined in Line 20 of the ex2.jsp page. This redefined value can be used in Line 15 of the ex2.jsp page again and referenced in Line 27 of the ex3.jsp page according to user navigation. From Fig. 7, we can derive the corresponding def-use chains <20, 15> and <20, 27> that cover the path from node 20 via nodes L and C to nodes 15 and 27. Notice that the definition of _session_ in Line 20 of ex2.jsp can reach to Line 27 of ex3.jsp via the path 20-21-Exit-(L-C-Entry-12-13-14-15-16-17-18-19-20-21-Exit)*-L-C-Entry-25-26-27.

Likewise, the value of _userBalance_ can be redefined in Line 30 of the ex3.jsp page and then referenced in Line 15 of the ex2.jsp page. The corresponding def-use chain <30, 15> can also be obtained from Fig. 7. Table 5 shows the def-use chains for the session attributes _userBalance_ and _userBalance_ and their corresponding aliases among the ex1.jsp, ex2.jsp, and ex3.jsp pages.

It should be pointed out that not all of the def-use chains computed from the SCFG are feasible since some data flow paths (or possible navigation paths) depicted in the SCFG could be infeasible. The infeasible paths can be addressed from the syntax and semantic perspectives of the JSP pages. From the syntax point of view, for example, consider a definition of _userBalance_ in node 20 in Fig. 6. By analyzing the ex2.jsp page in Fig. 6, we can find that the definition cannot reach node 27 via the path 20-21-Exit-(L-C-Entry-12-13-14-15-16-17-18-19-20-21-Exit)*-L-C-Entry-25-26-27 although this path can be derived from the SCFG. The reason is that the exp2.jsp page has two execution paths. However, the execution path passing through node 20 can generate a client page only leading to ex2.jsp itself. To visit ex3.jsp, the other execution path via node 23 must be covered. This means that the node 23 must be included in the data flow path for the definition of _userBalance_ in node 20 to reach its use in node 27.

Moreover, consider the def-use chain <30, 27> that can be derived from the SCFG in Fig. 6 for the attribute _userBalance_. This def-use chain is also infeasible because the client page generated by ex3.jsp can only link to the ex2.jsp page. As a result, the path that covers 30-31-32-Exit-L-C-Entry-25-26-27 is syntax infeasible. In addition, from the semantic point of view, by carefully analyzing the ex2.jsp and ex3.jsp pages we can find that the condition _acctBalance>=100_ in Line 17 of ex2.jsp can always be true when users navigate from ex3.jsp to ex2.jsp. This indicates that the path covering node 20 will be executed. Thus, the definition of _userBalance_ in Line 30 of ex3.jsp cannot be referenced in Line 27 of ex3.jsp without redefinition in Line 20 of ex2.jsp.

Note that it can be intractable to check if a path of the SCFG is semantic infeasible. This is similar to decide whether an execution path is infeasible in the traditional programs. However, some syntax infeasible paths of the SCFG can be identified by analyzing the navigation relationship among the JSP pages involving the session object.

Table 5
The session def-use chains among the ex1.jsp, ex2.jsp, and ex3.jsp pages

<table>
<thead>
<tr>
<th>Attributes of the session object</th>
<th>Session def-use chains</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>userName</em></td>
<td>&lt;6, 13&gt;, &lt;6, 26&gt;</td>
</tr>
<tr>
<td><em>userBalance</em></td>
<td>&lt;7, 15&gt;, &lt;7, 27&gt;, &lt;20, 15&gt;, &lt;20, 27&gt;, &lt;30, 15&gt;</td>
</tr>
<tr>
<td><em>name</em></td>
<td>&lt;13, 14&gt;, &lt;13, 26&gt;</td>
</tr>
<tr>
<td><em>tempBal</em></td>
<td>&lt;16, 19&gt;, &lt;16, 20&gt;, &lt;28, 29&gt;, &lt;28, 30&gt;</td>
</tr>
<tr>
<td><em>acctBalance</em></td>
<td>&lt;16, 17,23&gt;, &lt;16, 17,18&gt;, &lt;16, 18&gt;, &lt;16, 19&gt;, &lt;16, 19&gt;</td>
</tr>
</tbody>
</table>
Table 6
The Def-Use relationship among the ex1.jsp, ex2.jsp, and ex3.jsp pages

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Def-page (D)</th>
<th>Use-page (U)</th>
<th>Def-use page pair (D, U)</th>
</tr>
</thead>
<tbody>
<tr>
<td>userName</td>
<td>ex1.jsp</td>
<td>ex2.jsp, ex3.jsp</td>
<td>(ex1.jsp, ex2.jsp), (ex1.jsp, ex3.jsp)</td>
</tr>
<tr>
<td>userBalance</td>
<td>ex1.jsp, ex2.jsp, ex3.jsp</td>
<td>ex2.jsp, ex3.jsp</td>
<td>(ex1.jsp, ex2.jsp), (ex1.jsp, ex3.jsp), (ex2.jsp, ex2.jsp), (ex2.jsp, ex3.jsp), (ex3.jsp, ex2.jsp), (ex3.jsp, ex3.jsp)</td>
</tr>
</tbody>
</table>

For illustration, consider the ex1.jsp, ex2.jsp, and ex3.jsp pages in Fig. 6. Table 6 lists the possible definition-use relationships among ex1.jsp, ex2.jsp, and ex3.jsp obtained from their associated SCFG. In Table 6, the def-page D is a set of pages where an attribute of interest can be defined. The use-page U is a set of pages where the attribute of interest can be used. The def-use page pair P = (D, U) is a set ordered pairs (d, u), where d is a page from D and u is a page from U. Thus, P suggests the pairs of starting and terminating pages of possible data flow paths for an attribute.

For instance, there exist six possible def-use page pairs for the attribute userBalance. Specifically, the def-use pairs indicate that a data flow path from ex3.jsp to itself is possible. However, as shown in Fig. 8, the navigation relationship among the ex1.jsp, ex2.jsp, and ex3.jsp suggests that the data flow path from ex3.jsp to itself cannot be feasible unless the path passes through the ex2.jsp page. As a result, the path 30-31-32-Exit-L-C-Entry-25-26-27 can be identified to be infeasible since the path does not pass through the ex2.jsp page.

The navigation relationship depicted in Fig. 8 can help to remove some syntax infeasible data flow paths in the SCFG. However, Fig. 8 shows only the static navigation information and may be inadequate for identifying all syntax infeasible paths. One possible way to cope with this problem is to take into account the dynamic navigation behavior and require both the static and dynamic analysis of Web applications.

The efforts required to compute the data flow information of JSP-based Web applications can increase significantly when the attribute aliases of implicit objects and the sessional data flow artifacts are considered. In practice, testers would focus on some selected JSP pages that process the data of interest and use the test models discussed in Section 4 to generate def-use chains. Based on the test data adequacy criteria [19,20], tester can select appropriate def-use chains and derive test cases to uncover the data anomalies of JSP-based Web applications.

6. Conclusions

This paper has introduced an approach to support data flow analyzing and testing of JSP-based Web applications. In the approach, the data flow test artifacts posed by the implicit objects and action tags specific to JSP pages are identified and described. Test models are proposed to represent the JSP data flow test artifacts pages with the considerations of the control flow characteristics of various JSP implicit objects and action tags. The possible situations where JSP implicit objects and action tags can cause interprocedural data flow are also highlighted.

Moreover, we present a test model to capture the data interactions across different JSP pages introduced by the session object. Such data interactions can occur according to user navigation paths rather than occur based on the control flow of the JSP pages. In order to compute the data flow artifacts of JSP pages, we specify the definitions and uses for the attributes and Java objects involving implicit objects and action tags. The data flow information of JSP page for script variables, implicit objects, and action tags are then computed from the intraprocedural, interprocedural, and sessional perspectives.

Furthermore, we point out that there can exist infeasible paths across JSP pages when computing the sessional data flow artifacts. Two examples are employed to illustrate the infeasible data flow from the syntax and semantic perspectives of the JSP pages. A method is suggested to identify possible infeasible paths so that some infeasible data flow artifacts for the session object can be eliminated.

The development of a prototype tool is currently in progress so as to automate the collection of the definition and use information for variables, implicit objects, and action tags and to facilitate the computation of the def-use chains for JSP pages. Further plans include an empirical study to evaluate the proposed approach for JSP pages and an investigation to integrate the data flow test artifacts of JSP pages and HTML pages together for supporting the data analysis and testing of JSP-based Web applications.

Acknowledgements

We thank the anonymous referees for their valuable comments to improve this paper.

References
