Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Automated Migration of List Based JSP Web Pages to AJAX

Jason Chu and Thomas Dean
Department of Electrical and Computer Engineering
Queen's University
{0jhpc,tom.dean}@queensu.ca

Abstract

AJAX is a web application programming technique that allows portions of a web page to be loaded dynamically, separate from other parts of the web page. This gives the user a much smoother experience when viewing the web page. This paper describes the process of converting a class of web pages from round-trip to AJAX.

1. Introduction

One of the perennial problems in software maintenance is updating existing systems to use new technologies. When new technologies are introduced, it is not always cost effective to migrate existing systems to the new technology. Automated transformations can reduce both the risk and the cost of the migration.

The world wide web started as a mechanism for sharing documents, but has evolved, providing a portable, platform independent user interface for applications. The ability to use JavaScript to communicate with the server independently of the browser get/ submit actions, with the addition of the functionality to generate and parse XML led to what is now called Asynchronous JavaScript and XML (AJAX) [10,16]. The main use of AJAX is to update a portion of the web page without reloading the entire page. This gives the user a smoother web browsing experience and, in some cases, also conserves bandwidth.

Migrating round trip legacy web applications to utilize AJAX is not a trivial task. In this paper, we show a set of source level transformations that automatically migrates a class of web pages to utilize AJAX. The class of web pages handled by our transformation are pages written in JSP that display a subset of some form of list. These pages usually contain links that select the next (or previous) page in the list and may also have links that jump to to specific pages of the list (i.e. links to page 2, 3, 4 and 5). An example of pages in this class are the catalogue pages in an e-commerce site (e.g. DVD action movies for a movie rental site). Another example is the search result pages for a search engine such as Google. Our transform produces a page in which the previous and next links use AJAX to load the next or previous section of the list without affecting the rest of the page. The jump links are also similarly transformed. Our transformations only represent part of a more complete migration to AJAX. We do not handle the merging of multiple JSP pages such as merging the form with the search results.

This paper consists of 7 sections. Section 2 gives an overview of the approach. Section 3 discusses the manual preparation of the system and Section 4 details the steps involved in extracting a web service that provides the appropriate data in XML format from the JSP source. Section 5 describes how the page can be modified to use the new web service. Section 6 discusses some preliminary results of our process. Related work is described in Section 7 and the paper is concluded in Section 8.

2. Approach

The dynamic portions of a web page are not necessarily limited to the table of data that is the target of our transformation. Other items in the page such as the navigation menu, headers and footers and bread crumbs may also be generated dynamically. Thus the first step in our process is a manual identification of the code that is to be migrated. The rest of the process is entirely automatic. Figure 1 shows the structure of our transformation process. There are two streams to

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.29

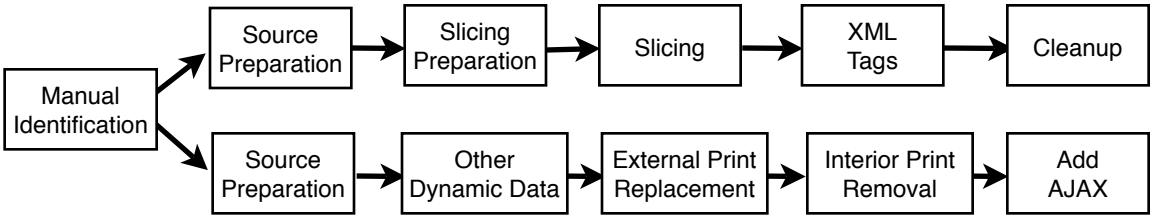

Figure 1. Transformation Process

the transformation, each of which starts with source preparation steps that are very similar to each other, but have some stream dependent details. The first stream (upper right of Figure 1) uses slicing to extract a simple Web service that generates the table data in XML format. The other stream (lower right) removes the generation of the list from the JSP source and replaces it with AJAX routines that call the service produced by the first transformation stream.

Most of the transformations are done using TXL[2] and the HTML/JavaScript/ASP grammars originally demonstrated in Synytskyy et al.[14,15] and modified for JSP by Xinzheng Li[6] and Andy Mao[6]. We extend the grammar to allow XML markup, specifically the manual markup used to identify the table to be transformed, as well as the markup used to implement the slicing algorithm.

3. Manual Identification

The user first identifies the primary section of code that is to be transformed using the XML tag `<ajax::mark>`. This is typically a loop that extracts and formats the data from a database query. Figure 2 shows such a markup. In the figure, a Java `while` loop has been marked. The loop iterates over the result set from a JDBC query (`rs.next()`), generating a single table row for each element. To simplify the figure, some of the details in this figure have been elided.

Other parts of the page may also depend on the contents of the table. Some examples are the page title in a heading at the top of the page, a page number and any next or previous links to navigate the query results. The `<ajax::annotate>` tag is used to identify this dynamic data. There are two minor changes in our markup from pure XML. The first is that we use square brackets to delimit attribute values to more easily manipulate them in TXL. The second is that we allow attributes on the close tag as well as the open tag. Figure 2 shows an example of the markup of the code that generates a previous link for a page. The *type* attribute of the annotation is `dlink`, which is used to indicate that the annotation may contain a dynamic link which must be modified to invoke the AJAX loading function as part of the page translation. As with Figure 2, some of the generated HTML text in Figure 3 has been elided for clarity.

```
<ajax::mark>
  while((iCounter<RecordsPerPage)&& rs.next()){
    getRecordToHash(rs,rsHash,aFields)
    String fldauthor = (String)rsHash.get(”mauthor”);
    String flddateentered = (String)rsHash.get(”mdateent”);
    String fldtopic = (String)rsHash.get(”mtopic”);
    out.println(”<tr>”);
    …
    out.println(”</tr>”);
    iCounter++;
  }
</ajax::mark>
```

Figure 2. Table markup.

```
<ajax::annotate>
if (iPage == 1){
  out.print("\n<a href=\"#\">…Previous…</a>");
}else{
  out.print("\n<a href=\""+sFileName+"?"+formParams+sSortParams
      +"FormMessagesPage="+(iPage-1)+"#Form\"> … Previous … </a>");
}
</ajax::annotate type=[dlink]>
```

Figure 3. Dynamic link markup

4. The XML Web Service

This section describes the extraction of a web service from the JSP page that generates the dynamic data in XML format.

4.1 Source Preparation

As in previous research using the same grammars, a sed (unix stream editor) script is used to normalize newlines, convert all commenting conventions to a single comment form and to convert any literal quotes in the HTML to the HTML friendly form.

Since the result of the transform will be code that generates an XML representation of the dynamic data, there is no need for any of the existing HTML from the page. In a similar manner to that used by Synytskyy et al[15], a TXL transform is used to extract all of the interesting elements. The transform redefines the interesting elements to include only the Java elements of the combined grammar. Thus all HTML elements are removed leaving only the scriptlets (<%…%>) and JSP expressions (<%=…%>). The JSP expressions are further normalized by converting them to explicit print statements. That is the JSP expression `<%=A%>` is converted to the Java statement `out.print(A)`.

When a JSP page is invoked by an application server, it is transformed by the server into a servlet. In this transformation, the code in the page is inserted into the `getRequest()` method. The developer can declare other methods and instance fields that can be shared between the methods. To simplify the slicing transform, these declarations are moved to the start of the page. The individual scriplets are also merged to produce a single scriptlet containing all of the executable Java code that will end up in the `getRequest()` method.

The last step is to convert any string concatenations in output statements that are within the `<ajax::mark>` tags into multiple statements, each referencing a single string expression. Output statements outside of this tag, including those manually identified with the `<ajax:: annotate>` are not modified. Instead they are annotated with a unique identifier that will be used as the XML tag for the data that they produce. This unique identifier is formed by concatenating all of the identifiers in the expression and an integer to make the identifier unique.

4.2 Slicing Preparation

Before we can slice, we must first identify the slicing criteria. Slicing criteria are a set of statements and a set of variables in those statements[17]. In this case, any statement that uses a Java variable when writing output is a slicing criteria candidate. We start by assuming that all such statements within the manual markup (`<ajax::mark>` and `<ajax:: annotate>` tags) and the variables referenced are slicing criteria. Any variables that have constant values (i.e. are assigned single literal values once) are removed, as are the output statements that use these variables.

We use the `<ajax:: annotate>` tag with parameters to identify the slicing criteria and, as will be shown in the next subsection, implement the slicing algorithm. Our transform starts by annotating the remaining output statements in the manually identified segments of code with the `<ajax::annotate>` tag, but with a type attribute of `data`. The keyword attribute of the annotation gives the set of variables for the slicing criteria. Figure 4 shows the annotation of one such output statement.

4.3 Slicing

We us a standard backward static slicing algorithm that is implemented as a set of source level transfor-

```
<ajax::annotate>
  out.print(fldField1) ;
</ajax::annotate type=[data]
             keyword=[fldField1]>
```

Figure 4. Identifying slicing criteria

mations. The algorithm is a modification of the one previously done for the Tiny Imperative Language by Cordy[3]. The algorithm propagates `<ajax:: annotate type=[data]>` tags, adjusting the contents of the `keyword` attribute to maintain the set of active variables. Elements of the transform handle all of the standard cases including loops and conditional statements. One significant difference is that the algorithm starts from multiple slicing criteria (one for each output statement in a manually marked segment), and as a consequence the transform must merge the variable sets appropriately when one trace runs into another. The annotation is only applied to those statements that are elements of the slice.

After the annotation of the executable is complete, any global variables or instance fields referred to by the annotated code are also annotated. After this transform the slice is still embedded in the original code as annotated statements. Any unannotated code is removed, as are the all of the annotations other than the annotations on the statements that print the dynamic data. The result of the slicing algorithm is an executable JSP page whose output contains only the HTML text that was generated by the output statements marked code segments of the original JSP page.

4.4 XML Tags

At this point in time, we have the core of JSP page that will generate only the dynamically generated values in the identified print statements. When executed, the results look like a stream of data with no breaks or space of any sort between the data items. The next step is to insert statements to generate the appropriate XML tags.

This starts with the insertion of the two JSP code elements shown in Figure 5. The first of these includes several utility functions that are used by the transformed code. The second sets the content type of the page to XML so that the browser AJAX routines will properly parse into a document object model (DOM) instance.

```
<%@ include file="AJAXCore.jsp" %>
<%@ page contentType="text/xml" %>
```

Figure 5. XML generation support

The general structure of the resulting XML is given with three tags, `<ajaxdata>`, `<ajaxlist>` and `<axaxitem>`. The top level of the XML document is given by a single `<ajaxdata>` tag, which contains both the table data (`<ajax::mark>`) and any other dynamic data that was manually identified (`<ajax:: annotate>`). The table data is identified using the `<ajaxlist>` tag, and each row of the table is identified with an `<ajaxitem>` tag.

These tags are extended with tags that are generated for each of the generated values. The names of the tags are dervied from the names of identifiers in the expressions. At the current time, we do not generate a DTD to allow the browser to perform conformity checks on the resulting XML.

Print statements to add the `<ajaxdata>` tag to the start and end of the result. Print statements are also added to enclose the list of items with `<ajaxlist>` tags and each line of the table with `<ajaxitem>` tags. Code is also inserted around each print statement that brackets the generated value with an XML tag as described above.

Calls to two utility routines, `AJAXLinkConvert` and `HTMLFilter` are also inserted into the code. The first routine is inserted in any print statement that was manually marked as emitting HTML links. It scans the generated link for references to the current page and replaces it by an appropriate AJAX call.

Because the dynamic code may still generate HTML code such as font tags for custom coloring of text, the `HTMLFilter` routine is used to ensure that such values are properly converted to XML safe values. For example angle brackets are converted to '<' and '>'. Figure 6 shows the results of the loop that was originally shown in Figure 2.

4.5 Cleanup

The last part to this phase of the transformation is to clean up the resulting code, including reverting comments to their normal form. Figure 7 shows results generated by the final JSP page.

As mentioned previously the structure of the table is identified by the `<ajaxlist>` and `<ajaxitem>`

```
out.print("<ajaxlist>");
while((iCounter<RecordsPerPage)&&rs.next())
{
  out.print("<ajaxitem>");
  getRecordToHash(rs,rsHash,aFields);
  String fldauthor = (String)rsHash.get("mauthor");
  String flddateentered = (String)rsHash.get("mdateentered");
  String fldtopic = (String)rsHash.get("mtopic");
  out.print("<toURL-rsHash-get-mmessageid>");
  out.print(HTMLFilter(toURL((String)rsHash.get("mmessageid"))));
  out.print("</toURL-rsHash-get-mmessageid>");
  out.print("<toHTML-fldtopic>");
  out.print(HTMLFilter(toHTML(fldtopic)));
  out.print("</toHTML-fldtopic>");
  out.print("<toHTML-fldauthor>");
  out.print(HTMLFilter(toHTML(fldauthor)));
  out.print("</toHTML-fldauthor>");
  out.print("<toHTML-flddateentered>");
  out.print(HTMLFilter(toHTML(flddateentered)));
  out.print("</toHTML-flddateentered>");
  iCounter++;
  out.print("</ajaxitem>");
}
out.print("</ajaxlist>");
```

Figure 6. Modified JSP dynamic generation code

```
<ajaxdata>
 <ajaxlist>
  <ajaxitem>
   <toURL-rsHash-get-m_message_id>20</toURL-rsHash-get-m_message_id>
   <toHTML-fldtopic>test</toHTML-fldtopic>
   <toHTML-fldauthor>test</toHTML-fldauthor>
   <toHTML-flddate_entered>2007-06-25 11:34:56</toHTML-flddate_entered>
  </ajaxitem>
  …
 </ajaxlist>
 <sFileNameformParamssSortParamsiPage0>
  &lt;a href="#"&gt;&lft;font … &gt;Previous&lt;/font&gt;%lt;/a&gt;
 </sFileNameformParamssSortParamsiPage0>
 <iPage1>[1]</iPage1>
 <sFilenameformParamssSortParamsiPage2>
   &lt;a href=javascript:getTable("indexXML.jsp?s_topic=&FormMessages_Page=
2#Form)"&gt; … Next … &lt;/a&gt;&lt;br&gt;
 </sFilenameformParamssSortParamsiPage2>
</ajaxdata>
```

Figure 7. Sample XML output of generated web service.

tags. In the example, there are three rows (the contents of the second and third row are elided in the figure to save space). The contents of the first row are "20", "test", "test" and "2007-06-25 11:34:56".

Outside of the table data, there are three values, given by the tags <sFileNameformParamss-SortParamsiPage0>, <iPage1> and <sFile-nameformParamssSortParamsiPage2>. The

```
if (iPage == 1) {
  <ajax::annotate>
  out.print("\n<a href=\"#\"> … Previous … </a>");
  </ajax::annotate tagname=[sFileNameformParamssSortParamsiPage0]
         type=[dlink2]>
}else{
  <ajax::annotate>
  out.print("\n<a href=\""+sFileName+"?"+formParams+sSortParams
      +"FormMessagesPage="+(iPage-1)+"#Form\">…Previous…</a>");
  </ajax::annotate tagname=[sFileNameformParamssSortParamsiPage0]
      type=[dlink2]>
}
```

Figure 8. Annotated other dynamic data print statements

```
if (iPage == 1) {
  out.print("<span id=\"sFileNameformParamssSortParamsiPage0\"></span>");
} else {
 out.print("<span id=\"sFileNameformParamssSortParamsiPage0\"></span>");
}
```

Figure 9. Annotated other dynamic data print statements

first and last contain anchors that refer to the previous and next pages of the table.

The final result of this sequence of transformations is a JSP page that returns the same data as the original page in XML format. Since the service is a slice of the original page, any side effects of the original page, such as server side session information, that are related to the list data are also included in the web service. As a result, the generated web service may not be fully compliant with the REST architecture. However, it provides a starting point for a more complete migration to a REST based architecture.

5. User Page Transformation

This section describes the migration of the JSP web page to use the web service described in Section 3.

5.1 Source Preparation

This step is very similar to the one described in Section 3. However, since the page is supposed to display the same contents, we leave all of the HTML elements in the code(they are not removed as they were in the previous section). There is one additional transformation in this step. All HTML code in the element containing the user markup(i.e. `<ajax::mark>`) is converted to Java print statements. This element may be a method, or it may be a scriptlet (<%…%>).

5.2 Other Dynamic Data Replacement

The first step in the transform of the web page is to deal with the sections of the page that were manually marked as extra elements to be migrated such as page number and menu elements that depended on the dynamic elements. During the source preparation step in both phases (XML service extraction and Page migration), additional annotation was introduced for each generated output that included an unique identifier. Figure 8 shows the result of this part of the normalization. The markup, originally shown in Figure 3 has been automatically copied to each of the output statements in the block of code and the attributes have been extended with an name generated from the names of the variables used in the output statements.

Each of the print statements is changed to generate an empty HTML span that uses the unique identifier as the value of the `id` as shown in Figure 9. The contents of the span will be inserted by JavaScript at runtime.

5.3 External Print Replacement

The general approach as illustrated in the previous section is to replace the manually marked sections with HTML span tags. However, in the case of the main section of data that was marked with an <ajax::mark> tag, there are some additional complexi-

```
"' <tr> '","'  <td> '","'<font face=\"arial\" size=\"2\">'",
"'<img src=images/Thread small.gif>'","' </font>'","'</td>'","'<td>'",
"'<a href=\"viewthread.jsp?'","'mid='","+toURL-rsHash-get-mmessageid",
"'&\"><font face=\"arial\" size=\"2\">'","+toHTML-fldtopic",
"'</font></a>'","'</td>'","'<td>'","'<font face=\"arial\" size=\"2\">'",
"+toHTML-fldauthor","' </font>'","'</td>'","'<td>'",
"'<fontface=\"arial\" size=\"2\">'","+toHTML-flddateentered",
"' </font>'","'</td>'","'</tr>'"
```

Figure 10. Table data list.

ties. The tag is used to identify the main loop that generates the data. This is typically inside of an HTML TABLE tag. Inserting a SPAN element inside of a table will not result in well-formed HTML. This step checks to see if the user markup is inside of a table tag. If so, the page is changed to generate a two JavaScript string constants. One for the HTML that extends from the `<TABLE>` tag to the `<ajax:: mark>` tag, and the other for the HTML that extends from the `</ajax::mark>` tag to the end of the table (`</ TABLE>`). If there is no table tags, this transform does nothing.

5.4 Interior Print Removal

There are two types of data that are generated within the user markup. Some of the data is static HTML, the other is the dynamic values that are now generated by the web service. This step generates the JavaScript that will create the same output for this region of the page as would be created by the original JSP. During the code normalization step, all embedded HTML within the user markup was converted to Java print statements.

In this transform, all of the print statements are removed, and the arguments are merged into a commas separated list of strings. The statements that generate dynamic data result in a string containing the unique identifier prefixed with the string concatenation operator ('+'). Static text (i.e. string or character literals) are a string containing the static data as a character literal. Any variables with static values (see section 3.3 Slicing preparation) are replaced by the constant values. Figure 10 shows such a list. This list is used by the transform described in the next section to create the JavaScript that will generate the full HTML and dynamically insert it into the page.

5.5 Adding Ajax Support

The last steps involve the final support for AJAX. The first transform in this step is to replace the marked up code with a single call to a helper routine, `ajaxExData`. This routine takes two parameters, the JSP output stream, `out`, and the URI of the page, which is obtained with the expression `request.getRequestURI()`. This helper routine will genenerate four elements into the page when it is called:

1) The empty SPAN tag that will contain the data.
2) Two supporting JavaScript files. The first contains the generic methods to support AJAX, and the second is a custom generated JavaScript function.
3) If a TABLE tag was found, then the constant strings generated in Section 4.3
4) A call to the JavaScript support library to fill in the empty SPANs when the page is first loaded.

The custom generated JavaScript function, named `tableMerge`, is created from the list of print statement parameters (e.g. Figure 10) created by the transform described in Section 5.4. It iterates over the each of the `ajaxitem` elements in the `ajaxlist` element of the DOM instance constructed by the browser when the web service is invoked. The body of the loop contains a sequence of string concatenations generated from the list of strings building up a row for the table. References to ajax tags are changed into calls to the `getElementByTagName` method of the DOM object. If the transform in Section 5.3 found a TABLE tag, then the JavaScript strings that were inserted by step 3 above are also included in the concatenation. Figure 11 shows part of the function generated from the string list in Figure 10.

After the core method included in step 2 above calls the custom function, it then checks the top level of the DOM object for any tags other than the `ajaxitem` tags. If any are found, it converts the text back to

```
function tableMerge(ajaxlist)
{
  var strTable = "";
  strTable += ajaxPreCont; // <TABLE> to <ajax::mark>
  for(varrowIndex=0;rowIndex<ajaxlist.childNodes.length; rowIndex++){
    var row = ajaxlist.childNodes.item(rowIndex);
    strTable+=" <tr> ";
    strTable += " <td> ";
    …
    if(row.getElementsByTagName("toURL-rsHash-get-mmessageid"[0].firstChild
                                                                  !=null)
      strTabl +=row.getElementsByTagName("toURL-rsHash-get-mmessageid"[0]
                                                  .firstChild.nodeValue;
    …
    strTable+="</td>";
    strTable+="</tr>";
  }
  strTable+=ajaxPostCont; // </ajax::mark> to </TABLE>
  return(strTable);
}
```

Figure 11. Custom JavaScript function

HTML and writes the contents of these tags to the HTML SPAN elements with the same name.

6. Preliminary Results

The transformation has been used to transform four open source web applications. Three of the web applications are open source applications from the website GotoCode.com. They were created using CodeCharge by YesSoftware. CodeCharge is an integrated development environment that provides an interface for attaching snippits of Java code to HTML form elements, authoring JSP pages and automatically handles some of the low level details such generating code for database connectivity.

The fourth application is an open source blogging web application, JSPBLOG[1], available on the Sourceforge application hosting site. All of the examples used in figures in the previous sections are taken the forum web application from GotoCode.com. Figure 12 shows the web page generated by the forum application.

In all of but one of the cases, the rendering of the final page is identical to the original page. The one exception(JSPBLOG) introduced one extra space after a single quote in a 1st level heading as a result of the default formatting of the TXL engine. All of the applications performed identically to the untransformed versions (except that the lists contents loaded without reloading the rest of the page).

7. Related Work

Automated migration of web applications is not new. Hassan et al. [4] used the island grammar technique to migrate web applications written in the ASP framework to NSP (Netscape Server Pages) framework. Ping et al.[11] use a transformation approach to migrate IBM Net.Data applications to the enterprise Java environment. Lau et al[5] demonstrated a migration from IBM Net.Commerce to Websphere Commerce Suite. All of these transformations translate the server side of the application. Our transformation changes both the server side and the client side.

There has also been recent research on converting existing applications to use AJAX. Puder [12] proposed a migration framework that allows AJAX applications to be written in Java. The application is first written as a Java desktop application and compiled. The generated .class file is then translated to JavaScript using XMLVM as an intermediate language. AJAX is used to transfer the JavaScript application from the server to the user.

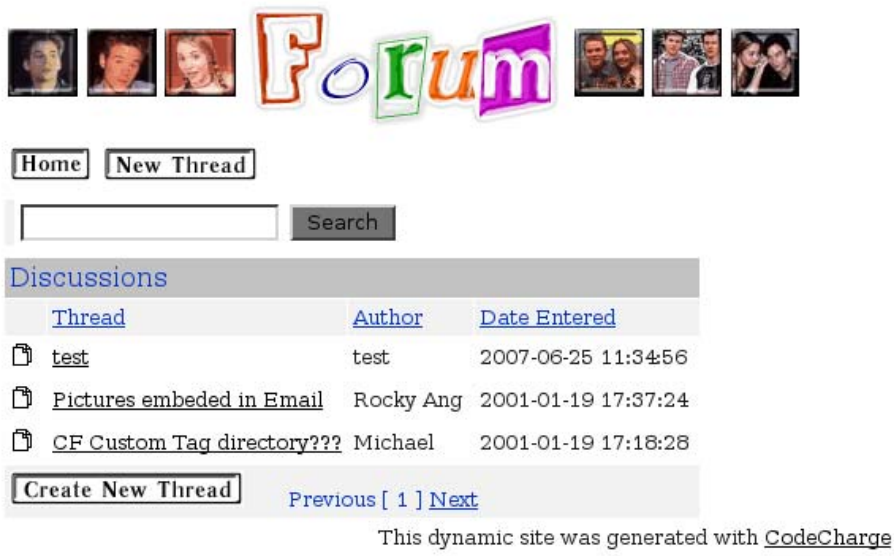

Figure 12. Round trip forum

Mesbah et al. [9] presented an approach to migrate multi-paged web applications to single-paged AJAX interfaces. They have implemented a tool, called RETJAX, which identifies web elements which are candidates for AJAX transformation. The approach presented does not produce an AJAX enabled product, but serves as a starting point to the transformation process. Their tool identifies potential candidates among multiple pages, ours focusses on the transformation of a single page.

Slicing has also been previously demonstrated for web applications. Ricca et al.[13] used slicing to demonstrate relationships between web pages. Lu et al. [18] applied slicing to web applications for the purpose of testing.

Previous work at our group has also used the unified grammar and TXL to transform web pages. This has included the conversion of simple JSP to custom tags [19,20], and the conversion of TABLE tags to DIV tags and cascading style sheets[8].

8. Future work and Conclusions

In this paper we have shown a set of source transforms that can migrate a class of round-trip JSP web page to AJAX. The class of pages are those pages that display a portion of a list with links to display other portions of the list. This involves extracting a web service and transforming the web page to use the service. The resulting web service is a JSP page that takes the same parameters as the original page and returns the data of the list in XML format. The transformations have been tested against several open source web applications, which functioned identically with the intended difference that navigating the list results did not require page reloads. However, the transforms have not been tested against any commercial applications.

The transformations do not handle the merging of JSP pages, and this is a clear area for future work. Although Mesbah et al.[9] do not provide an automated migration, they provide a technique for identifying candidates to be merged. This may provide an appropriate starting point for a more sophisticated transformation.

We have not yet tested pages that require secure access. Intuitively, the process should work since the generation of the data depends on the security checks, and therefore the security checks should be included in the slice that generates the XML. The JavaScript support routines will have to be adapted to deal with security failures during AJAX loads. This remains another area for future work

The implementation of our transform is specific to JSP. However, since the web service that is generated is a slice of the original page that has been modified to generate XML instead of HTML, the technique should

be adaptable to other embedded web application languages such ASP or PHP.

References

[1] Cguru. Jspblog from Sourceforge,net , http://sourceforge.net/projects/jspblog/ last accessed, April 2008.

[2] J. Cordy, "The TXL Source Transformation Language", *Science of Computer Programming*, 61(3), August 2006, pp. 190–210.

[3] James Cordy, *Backward slicing using TXL*, 2007. http://www.program-transformation.org/Sts/BackwardSlicingUsingTXL, last accessed February, 2008.

[4] A.E. Hassan, and R.C. Holt, "Migrating Web Frameworks Using Water Transformations", *Proc. 27th Annual International Computer Software and Applications Conference(COMSAC 2003)*, 2003, pages 296–303.

[5] Terence C. Lau, Jianguo Lu, Erik Hedges, and Emily Xing, "Migrating E-commerce Database Applications to an Enterprise Java Environment", *Proc of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Canada, November 2001, page 68-78.

[6] Xinzheng Li, *Defining and visualizing web application slices*, M.Sc. Thesis, Queens University, 2004.

[7] Andy Mao, *Translating Table Layout to Cascading Style Sheets*. M.Sc. Thesis, Queen's University, 2007.

[8] A. Mao, J.R. Cordy, T.R. Dean, "Automated Conversion of Table-based Websites to Structured Stylesheets Using Table Recognition and Clone Detection", *Proc IBM Centers for Advanced Studies Conference 2007 (CASCON)*, Toronto, Canada, October 2007. pp. 12–26.

[9] Ali Mesbah and Arie van Deursen. "Migrating Multi-page Web Applications to Single-page AJAX Interfaces", *Proc of the 11th European Conference on Software Maintenance and Reengineering*, Amsterdam, the Netherlands, March 2007, pp. 181–190.

[10] Sun Microsystems. *Asynchronous Javascript Technology and XML (AJAX) with the Java platform*, 2006.

[11] Y. Ping, J. Lu, T. Lau, K. Kontogiannis, T. Tong, B. Yi, "Migration of Legacy Web Applications to Enterprise Java Environments: Net.Data to JSP Transformation", *Proc 13th IBM Centre for Advanced Studies Conference (CASCON 03)*, Toronto, Canada, October 2003. pp. 223–237.

[12] A. Puder, "A Code Migration Framework For AJAX Applications", *Proc of the Seventh IFIP International Conference on Distributed Applications and Interoperable Systems*, Paphos, Cyprus, June 2006, volume 4025, pp. 138–151.

[13] F. Ricca, and P. Tonella, "Web Application Slicing", *Proc. IEEE International Conference on Software Maintenance*, Florence, Italy, November, 2001, pp. 148–157.

[14] Nikita. Synytskyy, J.R. Cordy, and T. Dean "Resolution of Static Clones in Dynamic Web Pages", *Proc. Fifth IEEE International Workshop on Web Site Evolution*, Amsterdam Netherlands, September 2003, pp. 49–56.

[15] N. Synytskyy, J.R. Cordy, T. Dean, "Practical Language-Independent Detection of Near-Miss Clones", *Proc 13th IBM Centre for Advanced Studies Conference (CASCON 03)*, Toronto, Ontario, October 2004, pp. 29–40.

[16] W3C. *The Xmlhttprequest Object,* http://www.w3.org/TR/XMLHttpRequest/, last accessed April 2008.

[17] M. Weiser, "Program Slicing", In *Proc of the 5th International Conference on Software Engineering*, 1981 pages 439–449.

[18] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen, "Regression Testing for Web Applications Based on Slicing". *Proc. 27th Annual International Computer Software and Applications Conference COMPSAC 2003*, pages 652–656.

[19] S. Xu, and T. Dean, "Modernizing Javaserver Pages by Transformation", *Proc. Seventh IEEE International Symposium on Web Site Evolution (WSE 2005)*, Budapest, Hungary, September 2005, pp. 111–118.

[20] S. Xu and T. Dean, "Transforming Embedded Java Code Into Custom Tags", *Proc. Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, Budapest, Hungary, September 2005, pp. 173–182.