# A Lightweight Approach for Migrating Web Frameworks

**Ahmed E. Hassan and Richard C. Holt**
Software Architecture Group (SWAG)
Department of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa, holt}@plg.uwaterloo.ca

## ABSTRACT

Web application development frameworks, like the Java Server Pages framework (JSP), provide web applications with essential functions such as maintaining state information across the application and access control. In the fast paced world of web applications, new frameworks are introduced and old ones are updated frequently. A framework is chosen during the initial phases of the project. Hence, changing it to match the new requirements and demands is a cumbersome task.

We propose an approach (based on *Water Transformations*) to migrate web applications between various web development frameworks. This migration process preserves the structure of the code and the location of comments to facilitate future manual maintenance of the migrated code. Consequently, developers can move their applications to the framework that meets their current needs instead of being locked into their initial development framework. We give an example of using our approach to migrate a web application written using Active Server Pages (ASP) framework to Netscape Server Pages (NSP) framework.

## 1 INTRODUCTION

Typing "www.amazon.com" in a web browser's window, loads up the main page of Amazon's web site in the window. Following links to a couple of pages on the site and filling in a form with your credit card number, your card is charged and a week later your book arrives in the mail. By visiting Amazon's web site and ordering a book, you used Amazon's book purchasing web application, a distributed web application with a convenient interface delivered through a web browser. With the advent of the Internet, a new type of application has emerged - web applications. Web applications use the Internet's infrastructure. They are being developed and maintained everyday [6]. Reports indicate that web applications represent more than thirty percent of software applications across all industry sectors and this number is expected to grow as the web gains popularity [5].

Using a web browser as a client for a software application instead of a specialized client poses many challenges for developers. One of these challenges, called the session management problems, arises from the difficulty of identifying page requests with particular users. This problem arises from the use of the HTTP, a stateless protocol. The web browser connects to the web server and requests a page using HTTP. Once the page is served, the connection between the browser and the server is terminated. A new connection is established for each new page requested. For the server, each request is considered as an independent one. The server cannot determine if the requesting client is the same as a previous client or if it is a new client. This technique permits servers to handle a large number of clients, as the clients consume server resources only when they are requesting a page. A web application page may need to identify the user requesting a page as being the same user that had requested previous pages or a new user. For example, a page may need to determine if a user has already provided a username and password or if the page should ask the user to login. To summarize, the stateless nature of the HTTP protocol makes it difficult for a web application to recognize if two requests have originated from the same user during the same session.

A number of frameworks have been introduced to solve the session management problem and other related problems associated with the development of web applications. Each framework provides support for essential functions to expedite the development of web applications. Java Server Pages (JSP), Netscape Server Pages (NSP), Allaire Cold Fusion (CF) and Active Server Pages (ASP) are some of the most common frameworks in web application development.

Because of the fast pace of the industry and its immaturity, current web application development emphasizes implementation productivity with little concern about maintenance and evolution of the applications [13]. The choice of a framework is done early in the life of a project and too often with little thought about the future impact of such choice. Once a web application is built on top of a framework, migrating it to a new framework to meet the new demands is a challenging task, especially with tight schedules. Some frameworks provide a rich set of libraries to speed up the initial product release, but they usually suffer from many performance bottlenecks which hamper the application's evolution. The embedded business logic and requirements in the code of the application discourage companies from rewriting it from scratch to make use of new web frameworks. Also such rewrite would require a great amount of time and effort.

Therefore, applications need to be migrated to new frameworks when their current framework is no longer supported or if it lacks new technology features (such as the recent need for migration from ASP to ASP.Net - the latest version of Active Server Pages).

In this paper, we propose an approach to migrate from one web development framework to another, in particular we show an example of migrating a web application from the ASP to the NSP framework. The approach aims to ease future maintenance of the migrated code by preserving the structure of the code and the location of the comments in the migrated code. The approach is largely automated and robust. It marks code segments that it fails to migrate so developers would review the marked segments and manually migrate them. The approach – *Water Transformations* is an extension of *Island Grammars* [20], which have been previously used for reverse engineering, software understanding, and impact analysis of traditional [24, 26] and web based legacy software systems [15].

Through our approach, developers can move their applications to the framework that meets their current demands instead of being locked in their initial development framework.

**Organization of Paper**

The rest of this paper is organized as follows. Section 2 describes web application frameworks. Section 3 presents our migration process, introduces the concept of *Water Transformations* and explains how we use them to ease the migration of multi-lingual software systems. Section 4 gives a detailed example of the migration using the formal approach presented in the previous section. Section 5 explains the technique and tools used to automate the code migration part of the process. Section 6 compares our work with other research projects and industrial tools which address similar problems. Section 7 draws conclusions from our work and proposes future directions.

## 2 WEB APPLICATION FRAMEWORKS

A web application framework provides support for the essential functions pertaining to the development of a web application, notably programming language support and a set of built-in objects (an object model).

A Web application is composed of active pages and static pages. Static pages are written in pure HTML. Active pages are like static pages but they contain *active* control code written in languages such as VBScript, Java, and JavaScript. The HTML tags are used to layout the data on the screen, and the control code uses the functionalities offered by the framework to customize the web page. Figure 1 shows an example of an active page written in VBScript.

When the client requests a static page, the server returns the page without modification. When the client requests an active page, the server preprocesses it. The control code is executed and the result of the execution is merged with the static content and returned to the requesting browser. The example shown in Figure 1 will appear as a page entitled `Main Page`, when viewed in a browser. The page will either welcome the logged-in user or ask the user to log into the system.

```
<HTML><TITLE>Main Page</TITLE>
<% If Session("LoggedIn") Then 'User already logged in %>
Welcome
<% Response.Write(Session("username")) %>
to your account, to display balance
     <A HREF="balance.asp">click here</A>
<% Else 'User needs to login %>
Please login first, <A HREF="login.asp">click here</A>
<% End If %>
</HTML>
```

**Figure 1:** Example ASP File

Whereas web frameworks permit the embedding of control code in HTML, other technologies such as CGI, ISAPI, NSAPI, and Java Servlet take the reverse approach. Active pages written in such technologies are regular programs which use print statements (such as *printf("..")*) scattered through out the code to output the HTML code. We are not concerned with the migration of web applications developed using such technologies, as they are similar to regular application and can be migrated using traditional migration approaches. Instead, we focus on migrating web frameworks where the intermixing of languages (HTML and control language) complicates the adoption of traditional language migration approaches.

### PROGRAMMING LANGUAGES FOR WEB FRAMEWORKS

Each web framework has one or several programming languages which are used to express the control sections of active pages. JavaScript, VBScript, ColdFusion, and Java are the most commonly used programming languages.

The Active Server Page (ASP) framework uses VBScript, which is a subset of the Visual Basic language developed by Microsoft Corporation. It is a scripting language that is interpreted and runs primarily on Microsoft Windows. It is a weakly typed language with only one data type (`Variant`). VBScript is an object-based language that permits developers to specify objects (in terms of methods and properties) but does not support other object-oriented features such as inheritance. No extensive reference manual exists that describes the language in detail. Therefore to develop a migration tool, we examined many systems and code samples written in VBScript so we could infer the language's structure and syntax.

The Netscape Server Pages (NSP) framework uses JavaScript, also called ECMAScript [12], which is an object-based scripting language and is not a subset of any other language. JavaScript shares many similarities with object-oriented languages such as the Java and C++ languages.

New types cannot be defined and it has many built-in `data types` such as `String`, `Number`, `Boolean`, `Object`, `Array`, `Null`, and `Undefined`. The language is loosely typed; the data type of variables do not need to be defined ahead of time and conversion between the different data types is done automatically without the need of a cast operator. The data type of a variable is based on the value of the variable at run-time. The language is developed under the control of the ECMA [11] standardization organization. JavaScript is the name of Netscape Corp.'s implementation of the ECMAScript language whereas JScript is Microsoft Corp.'s name. Both implementations are supersets of the ECMAScript language, since they provide extensions specific to the implementing company. The extensions add many built-in objects and provide mechanisms for the language to interact more easily with other components in the NSP framework. The ECMAScript language can be interpreted or compiled.

```
<HTML>
<CFOUTPUT>Hello World</CFOUTPUT>
</HTML>
```

**Figure 2:** "Hello World" Active Page Written in Cold Fusion

ColdFusion (CF) is another web development framework, developed by Macromedia Corp [1]. Whereas JSP and ASP are based on active code embedded in HTML, CF uses a specialized markup language called Cold Fusion Markup Language (CFML) which is embedded in HTML as well. CFML has tags which represent the control flow of a traditional programming language such as `<CFIF>` or `<CFELSE>`. Figure 2 shows a program written in the CFML markup language. Finally, Java Server Pages (JSP) framework permits the embedding of active code written in the Java programming language, a strongly typed object-oriented language which shares many similarities with the Modula-3 and C++ languages.

**A COMMON WEB OBJECT MODEL FOR WEB FRAMEWORKS**

Each language is supported by a set of built-in objects which are provided by each web application framework. These objects abstract the commonly needed functionalities in the development of a web application, such as access to the client's request or the maintenance of the client's state across multiple HTTP requests. Through studying JSP, NSP, ASP, and ColdFusion frameworks, we were able to define a set of common objects across all frameworks (refer to Table 1). Table 2 maps the various objects in our common object model and the built-in objects in the studied frameworks. Through this mapping we can migrate across the various framework built-in objects, as we shall explain later.

## 3 THE MIGRATION PROCESS

| Object Name | Purpose |
|---|---|
| Request | The Request object stores details related to the request originating from the web browser to the web application. The object contains information entered by a user in a form. The object can also be used to retrieve cookies stored on the client side. |
| Response | The Response object describes the result of the request sent from the browser. An active page would write information to the Response object. The information could set a cookie on the client. The information could specify details to be displayed in the browser. |
| Session | The Session object represents the session of a particular user. The session starts when the user's browser requests a page from a web site and terminates after a configured timeout period since the last page request. Moreover, a session can also be forcibly terminated when a user wants to logout from an application. A Session object is used to communicate information across active pages in a web application. |
| Application | The Application object represents a global space for the whole web application. A web application starts when the first page in an application is requested by a browser. It ends when the application server is terminated. The Application object permits the sharing of information between all active pages in a particular web application. An Application object may store a variable to count how many users have accessed the application since it started. Also it can be used to pool database connections for reuse by the various active pages. |
| Server | The Server object is a global server level object. It is shared among all web applications running on the application server. It stores common configuration properties and state information, such as the memory usage of the server, or the email address of the server administrator. |
| Error | The Error object records all errors that occurred during the interpretation of an active page by the server. Each active page has an Error object. |

**Table 1:** Common Web Object Model

| Common Objects | ASP Objects | NSP Objects | JSP Objects | CF Objects |
|---|---|---|---|---|
| **Request** | Request | Request | javax.servlet.ServletRequest | <CFHttpParam> |
| **Response** | Response | _[1] | javax.servlet.ServletResponse<br>javax.servlet.jsp.JspWriter | <CFOutput> |
| **Session** | Session | Client | javax.servlet.http.HttpSession | <CFApplication><br><CFCookie> |
| **Application** | Application | Project | javax.servlet.ServletContext | <CFApplication> |
| **Server** | Server | Server | javax.servlet.ServletContext<br>javax.servlet.ServletConfig | <CFRegistry> |
| **Error** | ASPError | Ssjs_onError | java.lang.Throwable | <CFError> |

1. NSP does not encapsulate Response related methods in their own object; instead they reside in the global namespace.

**Table 2:** ASP, NSP, JSP and CF Object Models Mapped to the Common Web Object Model

This section presents our process for migrating web application to a new framework. As described in the previous sections, web applications contain two types of pages: active pages, and static pages. Static pages do not contain any control code; thus they are framework independent and are not processed during our migration process. Instead, we focus on migrating active pages, which contain control code that is framework and language dependent.

Web applications range from simple static application (such as a personal web site, a home page) to sophisticated e-commerce ones (such as *Amazon.com*, *eBay.com*). Whereas a large number of tools have been proposed to assist in building web sites, these tools either focus on building static application with little control code in them or on building business applications that follow well defined business flows [29]. They fail to provide the flexibility and support needed to build large scale industrial level applications. Web developers tend to build and maintain such large applications without using such rapid development tools [13, 14].

To keep up with the changing needs of users, it is common for the migrated code to undergo repeated changes – adding new features, enhancing old ones, or fixing bugs. Thus an automated migration process which produces unreadable code with comments stripped out of the original application is not an acceptable solution as the migrated application will be manually maintained and evolved under the new framework. Furthermore, the user interface of the application, which is defined using the HTML code intermixed with the active code, should not be altered as the application is migrated to a new framework. The users of the application should not notice user interface changes.

We define two main objectives for our migration process:

1. The migration process should ensure that HTML code in the migrated application, remains in the same location relative to the control code in the original application. The user interface should not be affected by the use of a different programming language or a different development framework.

2. The migration process ensures that all comments in the migrated code remain in the same locations relative to the original source code, so developers can still easily maintain the migrated application manually.

**Island Grammars**

In contrast to regular applications written in a single programming language, web application contain a variety of languages intermixed inside each active page. This intermixing of control code, comments inside the control code, and HTML tremendously increases the complexity of parsing such pages. In our previous work [15], which concentrated on visualizing web applications, we dealt with the existence of multiple sections written in different programming languages inside of a single source file by choosing to extract only the entities which we are interested in visualizing.

Each processed file is viewed as an ocean of tokens – we define two types of tokens “interesting” and “uninteresting” tokens. We examine consecutive interesting tokens (islands of interest) using set of extractors based on grammars for each island of interest (control code islands, HTML islands, etc.). Each extractor processes the source file and locates the subsections (islands) of interest in the file. Once these islands are located, the appropriate parse is performed to extract the information. For example if we define the HTML code as being our interest island, then all the control code tokens become water that is ignored by our parsers. Looking at Figure 1, lines 1, 3, 5, 7, and 9 would be islands and the other lines would be water.

The use of island grammars speeds up the parsing process, as island grammars are much smaller than full language grammars and we do not need to maintain complex details about the full parse tree. Island grammars are as well more robust and can recover from simple syntax errors as they only specify in detail small sections of the language and not the whole language.

**Definition:** An **Island Grammar** is a grammar which contains two types of productions: a) productions which describe constructs of interest (*Islands*), and b) liberal productions which catch the remaining uninteresting sections (*Water*).

Formally, for a context free language (CFL) $L_G$, a context free grammar (CFG) $G$, such that $L(G) = L_G$ is defined as $G$ = ($V$, $\Sigma$, $S$, $P$ ), where

1. $V$ and $\Sigma$ are finite sets with $V \cap \Sigma = \emptyset$: $V$ is the set of variables or nonterminal symbols, and $\Sigma$ is the alphabet of terminal symbols (terminals)
2. $S \in V$ is the $start$ symbol; and
3. $P$ is a finite set of $productions$ or grammar rules of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

We define an *Island Grammar* $G_I = (V_I,\ \Sigma_I,\ S_I,\ P_I\ )$ for a set of constructs of interest $I \subseteq \Sigma^*$ such that $\forall\ i \in I$, $S \Rightarrow^* s_1 i s_2$, where $s_1$ and $s_2 \in \Sigma$ and $s_1 i s_2 \in L_G$. This definition ensures that the island grammar is more liberal than the original grammar $i.e.\ L(G) \subseteq L(G_I)$. Therefore $G_I$ can parse correctly text that is not valid under $G$. As the input of the migration process is valid under $G$, we do not have to be concerned about this fact. A more complete analysis of island grammars is available in [25].

#### Water Transformations

Whereas *Island Grammars* are used to perform lightweight understanding of large code bases, they are not capable of performing powerful language transformations/migrations. As the generated parse tree is restricted to the island (areas of interests), the transformations can only be done on very simple/local areas inside the islands of interests [24]. This limits the usefulness of Island grammars in migrating web applications; while keeping the HTML interface of the application and the code comments intact relative to the migrated code. To accomplish this task, we identified three different alternatives:

1. **Filtering Uninteresting Code:** Remove all the HTML and control code comments from an active page, perform the language/framework migrations, then re-insert the HTML and code comments. In [8] the authors propose a similar approach to handle comments during the migration of a legacy COBOL system. Their approach removes the comment from the legacy code, performs the migration, then re-inserts comments using a modified diff algorithm which compares the pre-migration code and the post-migration code; and attempts to re-insert the comments in the appropriate locations.

   We choose not to use this approach because the extensive intermixing between HTML code, control code, and comments in a single active page file would increase the complexity of the proposed diff algorithm.

2. **Grammar Extension:** An alternative approach is to extend the grammar of the control code language to support the intermixing the HTML code, comments, and control code. For example to migrate from JavaScript to VBScript, we would extend the JavaScript and VBScript language grammars to support the embedding of HTML code and comments in the generated parse tree. Furthermore, we would have to define rules to specify how these new comment and HTML tokens will be processed during the language transformation.

   We decided not to adopt this approach. The complexity of extending each programming language grammar to support such intermixing is too high, it would require a good understanding of the grammars of each language, and a good experience in crafting such complex grammars. Furthermore, this would increase the complexity of the grammars resulting in a slowdown of the migration process.

3. **Water Transformations:** We opted to adopt *Water Transformations*, as a new technique to migrate multilingual source code bases. The approach is more light weight. The complexity of developing the transformation technique is much simpler than the other two described approaches, since the approach unifies the various languages into one language.

We now explain *Water Transformations*. Given a file that has a large intermixing of different languages (comments, HTML, and control code), we define an *Island Language*. The island language is the language we are interested in migrating. For our case that would be the control code languages, such as JavaScript, or VBScript. Following the same analogy as *Island Grammars*, the rest of the tokens are considered as *Water*. We now define *Water Transformations*, which are code transformations that convert water tokens to special islands. These new special islands follow the syntax of the island language (in our case the control code language). Once we apply these *Water Transformations* to the input file, the result is a valid file in the language of the island language with no Water (uninteresting) tokens left.

We then perform the migration using the grammar of the island language. The migration can be performed by any technique described in the vast literature of language migration such as [4, 21, 22, 28, 30, 32, 33, 35].

After the migration is performed, the inverse of the water transformations is applied on the output to revert the special islands back to water. The final output is a migrated file with the location of the HTML code and source comments left constant relative to the location of the newly migrated control code.

**Definition:** We define the **Water Transformation** $T_W$, as the transformation that maps each set of consecutive tokens $(s_i)^*$ in the input file to $A_j$ ($i.e.$ $(s_i)^* : \xrightarrow{T_W} A_j$), where $\forall i$, $(s_i)^* \not\subseteq L(G_I)$; $\forall j$, $A_j \subseteq L(G_I)$; and $j \in \{1, ..., n\}$, where $n$ is the number of independent consecutive streams of tokens $(s_i)^*$ in the input file.

In simpler terms, each continuous stream of tokens that are not valid in the island language ($(s_i)^* \not\subseteq L(G_I)$) are mapped to $A_j$ which is valid in the *island language*. For example, during the migration of an active page in ASP to NSP we can transform each stream of continuous HTML code into a call to a dummy function called `HTMLCALL()`. Whereas the HTML code is not valid code in the VBScript language, `HTMLCALL()` is valid code in the VBScript langauge.

Using *Water Transformation*, the migration of a software system can be written formally as:

$T_W^{-1}(T_M(T_W(Input\_File)))$,

where $T_M$ is the migration transformation, which converts from one programming language to the other. $T_W^{-1}$, the *Inverse Water Transformation* is defined as $A_j \ : \xrightarrow{T_W^{-1}} T_M^{-1}(A_j) = (s_i)^*$. This formula can be extended to as many embedded languages as needed, a more general formula is:

$T_{W_x}^{-1}(....T_{W_1}^{-1}(T_M(T_{W_1}(....(T_{W_x}((Input\_File))))))).$

In the following section, we present a case study of the migration process and a more detailed example than the abstract examples presented in this section. We use the term *Preprocessing* to indicate the application of the *Water Transformations* and the term *Postprocessing* to indicate the application of the *Inverse Water Transformations*.

## 4 CASE STUDY: AN EXAMPLE OF THE MIGRATION PROCESS

Figure 3 shows an overview of the migration process. The migration process is divided into four main stages:

1. The *Preprocessing* stage removes the HTML code and comments from the active page. These are set aside to be later inserted into the migrated file in the (*Postprocessing* stage). This stage corresponds to the `rmHTML` and `rmComment` boxes in Figure 3.

2. The *Language transformation* stage translates the active code from the initial programming language to the programming language used by the target web application framework. The translation is carried out by a program written in TXL [34].

3. The *Object model mapping* stage transforms access to objects in the original application framework to the corresponding objects in the target framework. For example, we would map *Response.Write* to *Write* to migrate from the ASP to the NSP frameworks.

4. The *Postprocessing* stage reinserts the HTML code and comments that were removed in the *Preprocessing* stage. This corresponds to the `addHTML` and `addComment` boxes in Figure 3.

We elaborate these stages in the following subsections.

***Preprocessing Stage***

During the *Preprocessing* stage the active page is processed by two `Perl` scripts:

1. The first `Perl` script (`rmHTML`) processes the ASP file and replaces contiguous sections of HTML code with a call to dummy function named `HTMLCALL()` with a numbered parameter. The removed HTML code is stored in another file in an XML format. This XML file and the numbering is used later to reconstruct the full NSP file. Figure 4 shows the result of this action when processing the active page shown in Figure 1.

```
HTMLCALL(1);
If Session("LoggedIn") Then 'User already logged in
HTMLCALL(2);
Response.Write(Session("username"))
HTMLCALL(3);
Else 'User needs to login
HTMLCALL(4);
End If
HTMLCALL(5);
```

**Figure 4:** Example ASP File After `rmHTML`

2. The second `Perl` script (`rmComment`) processes the output of the first script; it removes each line of comments in the VBScript code and replaces it with a call to dummy function named `COMMENTCALL()` with a numbered parameter. The same technique used to reinsert the HTML code is used to reinsert the comment code. The scripts perform some massaging of the source code to reduce the complexity of the grammar used in the following stage to parse the VBScript file. For example, it adds a semicolon at the end of each line of VBScript and breaks all statements that are separated with a ":" into separate lines[1]. Figure 4, shows the results

[1]VBscript permits multiple statements on the same line if the they are separated by ":"

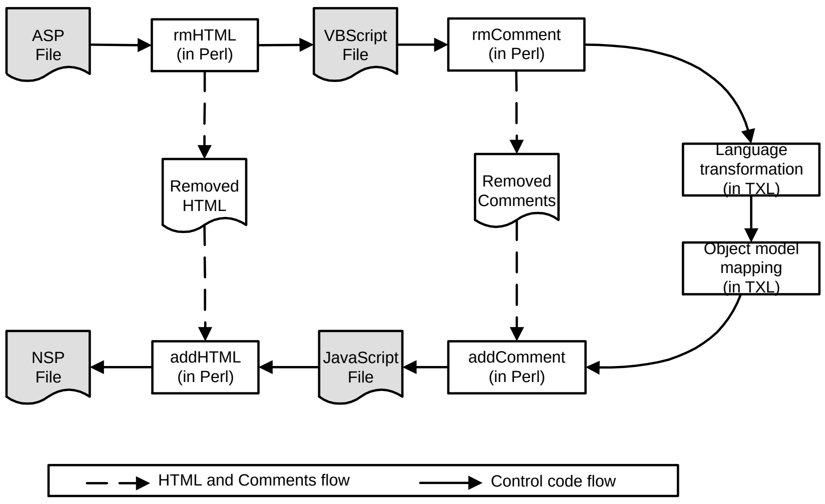

**Figure 3:** Migration Process

after running the `rmComment` Script.

```
1: HTMLCALL(1);
2: If Session("LoggedIn") Then; COMMENTCALL(1);
3: HTMLCALL(2);
4: Response.Write(Session("username"));
5: HTMLCALL(3);
6: Else; COMMENTCALL(2);
7: HTMLCALL(4);
8: End If ;
9: HTMLCALL(5);
```

**Figure 5:** ASP File After `rmComment`

At the end of the *Preprocessing* stage, the output file contains a valid VBScript program with no comments and with each line terminated by a semi-colon as shown in Figure 5.

As our migration process is performed on each file separately and code changes are only reflected locally in the processed file/code section, our migration process is simple and quite scalable. The location of code segments is not susceptible to large changes during migration, thus the location of HTML and comments remains constant relative to the active migrated code.

***Language Transformation and Object Model Mapping Stages***
Once the *Preprocessing* stage is completed, the output is processed by two programs written in the `TXL` [8, 34] programming language. The `TXL` programming language is a hybrid functional/rule-based language with unification, implied iteration and deep pattern matching. This language permits us to describe the transformation easily using abstract syntax nodes. Section 5 gives more details about these `TXL` programs. Figure 6 shows the results of the two TXL programs:

```
1: HTMLCALL(1);
2: If (Client.LoggedIn) { COMMENTCALL(1);
3: HTMLCALL(2);
4: Write(Client.username);
5: HTMLCALL(3);
6: } else { COMMENTCALL(2);
7: HTMLCALL(4);
8: }
9: HTMLCALL(5);
```

**Figure 6:** ASP File After `TXL` Processing

1. The *language transformation* program translates control code in one language to another, for example from VBScript code into JavaScript code. Each programming language has its own type system. Although VBScript and JavaScript are both weakly typed, VBScript has only one data type, while JavaScript has multiple data types. In the general case, we would need to determine the data types of each converted variable and map them to the corresponding data type in the target language. Fortunately, the JavaScript interpreter can determine the type of a variable at run-time based on its content. Thus, we do not need to perform type inference analysis. Instead the determination of the data types is left to the interpreter at run-time. Alternatively, to migrate an application which uses the ASP framework to one that uses the JSP framework, a type inference analysis is required as JSP uses a strongly typed language (Java). From our experience in studying web applications [14, 15, 16], we notice that a large percentage of the variables in active pages are of type `String`. During the migration to strictly typed language, the migration process may simply assume that all variables are of type `String` since it

is the most used variable type in web applications. The developer will need to examine the result of the migration to JSP and correct the output.

2. The *object model mapping* program examines object references to built-in objects provided by the ASP framework. The program transforms these references to ones to the corresponding built-in objects provided by the NSP framework (see Table 1). For example, during migration of an ASP file to a JSP file, each reference to the `Request` object is replaced with a reference to `javax.servlet.ServletRequest` object. Whereas the Table 1 shows high level mappings between object names, the `TXL` program specifies more accurately the mapping between the various frameworks. The program details the mapping at the level of method names and parameter order and type. In the cases, where no appropriate mapping exists a special token `<UNKNOWN>` is inserted and manual intervention is needed to correct the output. This `TXL` program could be extended by declaring new mappings between user defined or special objects.

The input to this stage is a file in a single programming language, namely VBScript, and the output is a file in the JavaScript langauge. The techniques used in this stage are similar to traditional program migration techniques and other techniques from the literature could be adopted. Using such techniques would not have been possible without the previous pre-processing stage which converts the multilingual active page into a single language page. The following stage undoes the Water transformations done in the preprocessing stage.

***Post-Processing Stage***
This is the last stage of the migration. Two `Perl` scripts process the output of the previous stage:

1. The `addComment` script reinserts source code comments that were removed by `rmComment` script in the pre-processing stage. It scans the input file and replaces each call to the place-holder function `COMMENTCALL()` with the corresponding comment line stored in the "*Removed Comment*" XML file.

2. The `addHTML` script reinserts the HTML code that was removed by `rmHTML` script in the pre-processing stage. Calls to the place-holder function `HTMLCALL()` are replaced with the corresponding HTML code.

Figure 7 shows the results of this last stage of the migration from ASP to NSP frameworks. The `<SERVER>` tag, shown in Figure 7, is used by the NSP framework to indicate control code in an active page. The active page shown in Figure 7 is equivalent to the active page shown in Figure 1.

```
<HTML> <TITLE>Main Page</TITLE>
<SERVER>If (Client.LoggedIn) { // User already logged in</SERVER>
Welcome
<SERVER> Write(Client.username) ; </SERVER>
to your account, to display balance
        <A HREF="balance.asp">click here</A>
 <SERVER> } else { 'User needs to login  </SERVER>
Please login first, <A HREF="login.asp">click here</A>
<SERVER> }  </SERVER>
</HTML>
```

**Figure 7:** Final NSP Active Page

## 5 TXL PROGRAMS

Our migration process uses the `TXL` language to express the transformation from one language to another. Since this transformation approach, as used to migrate web applications, is novel; we will present it in some detail in this section. The `TXL` language is useful for structural analysis and transformation of formal notations such as programming languages, specification languages, structured documents. It was used extensively during the Y2K analysis of COBOL/PL1 programs to repair date problems [10, 7]. It provides high level constructs to build a parse tree of an input program, to specify transformations for the parse tree using tree structure and textual pattern rules, and to emit the transformed parse tree. Each `TXL` program is composed of three sections:

- Parse tree specification for the input language.
- Parse tree extension for the target language.
- Transformation rules specification.

### PARSE TREE SPECIFICATION

The first section of a `TXL` program defines how the input file should be parsed and outlines the structure of the generated parse tree for the input. This is specified in a `BNF` like syntax.

The `TXL` code shown in Figure 8 defines the structure of the `IfThenElse` for the VBScript language. The keyword `opt` indicates that the node may not exist in the parse tree. The keyword `repeat` indicates that the node may be repeated multiple times. The `[stmts]` non-terminal specifies any valid VBScript statement. The `[expression]` non-terminal specifies all valid expressions in the VBScript language. Both definitions are given elsewhere and are omitted to improve the readability of the code snippet.

Figure 8 shows that an "`If`" statement may contain other statements. It may have multiple `[ElseIfBody]` clauses that may in turn contain more statements. It may have an optional `[ElseBody]` clause which may contain other statements. The generated parse tree would contain nodes corresponding to the `[expression]`, `[stmts]`, `[ElseBody]`, and `[ElseIfBody]` non-terminals. Figure 9 shows the generated `TXL` parse tree when the program in Figure 5 is processed by the TXL program.

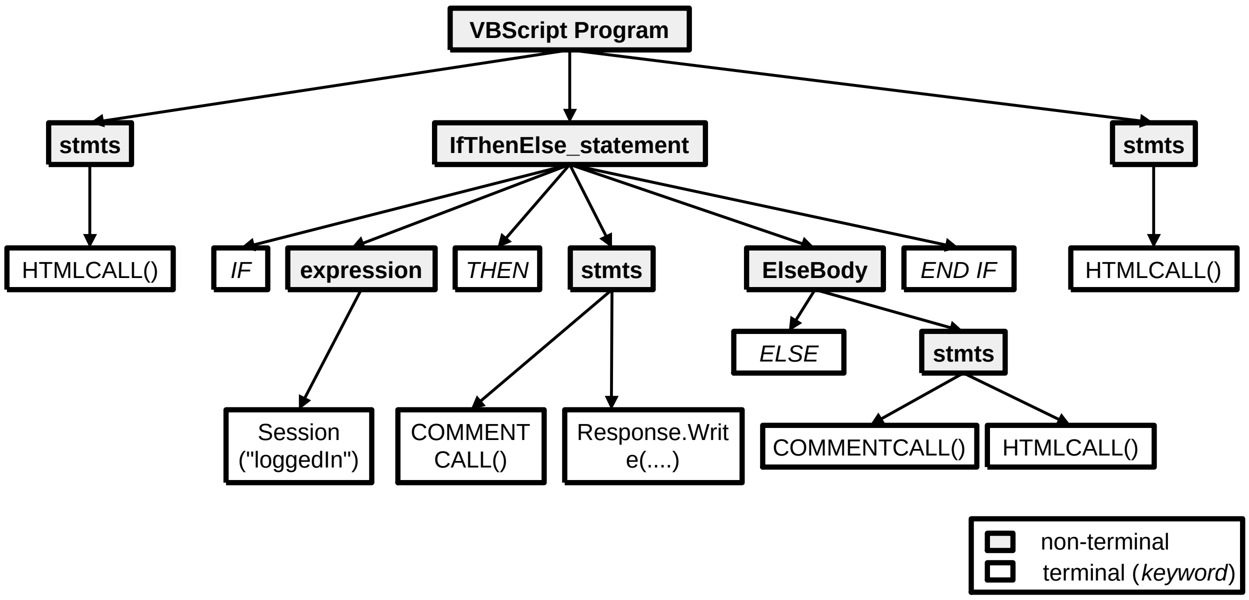

**Figure 9:** Parse Tree for Program Shown in Figure 5

```
define IfThenElse_statement
        IF [expression] THEN ;
                [stmts] [repeat ElseIfBody]
                [opt ElseBody]
        END IF;
end define

define ElseIfBody
        ELSEIF [expression] THEN ; [stmts]
end define

define ElseBody
        ELSE ; [stmts]
end define
```

**Figure 8:** Simplified Parse Tree Specification for The `IfThenElse` Statement in VBScript

**PARSE TREE EXTENSION**

The second section of a `TXL` program defines the syntax of the output in the target language. This is used to ensure that the parse tree is a valid parse tree during the execution of all the transformation listed in the `TXL` program. At the beginning of the transformation, the parse tree is a pure VBScript parse tree. After a couple of transformations, the parse tree is a mixture of a VBScript parse tree and a JavaScript parse tree. At the end - after applying all the transformations, it is a pure JavaScript parse tree. Figure 10 shows the extensions needed to the base VBScript grammar to specify the JavaScript grammar for the `IfThenElse` statement. In JavaScript the `If` statement condition expression is enclosed between parentheses. Curly braces are used to delimit the `If` statement instead of using the `END IF` keyword. Also the keyword `ELSEIF` is replaced with the keywords `ELSE` and `IF`.

```
redefine IfThenElse_statement
        ...
        | IF ([expression]) [opt { ]
                [opt stmts]
        [opt } ]
        [repeat ElseIfBody]
        [opt ElseBody]
end define

redefine ElseBody
        ...
        | ELSE [opt { ]
                [stmts]
        [opt } ]
end define

redefine ElseIfBody
        ...
        | ELSE IF ([expression]) [opt { ]
                [opt stmts]
        [opt } ]
end define
```

**Figure 10:** Extension of the `IfThenElse` Statement to Specify the JavaScript Language

**TRANSFORMATION RULES SPECIFICATION**

```
rule VBScript_ifthenelse_to_javascript
        replace [IfThenElse_statement]
                IF  EXP [expression] THEN ;
                        STMTS [stmts]
                EIB [repeat ElseIfBody]
                EB [opt ElseBody]
                END IF;
        by
                IF  (EXP) {
                        STMTS
                } EIB EB
end rule

rule VBScript_elsebody_to_javascript
        replace [ElseBody]
             ELSE ; STMTS [stmts]
        by
                ELSE {
                        STMTS
                }
end rule

rule VBScript_elseifbody_to_javascript
        replace [ElseIfBody]
             ELSEIF EXP [expression] THEN ;
                STMTS [stmts]
        by
                ELSE IF (EXP) {
                        STMTS
                }
end rule
```

**Figure 11:** Rules to transform the `IfThenElse` Statement from VBScript to JavaScript

The final section of a `TXL` program specifies the rules that will transform the VBScript program into a JavaScript program. Each `TXL` transformation rule must have, at least, one pattern to match and a replacement. A rule has this form:

```
    rule  name
         replace  [type]
              pattern
         by
              replacement
    end  rule
```

*Name* is an identifier, *type* is a non-terminal in the parse tree which the rule transforms, *pattern* is a pattern which the rule's input tree must match, and *replacement* replaces the matched *pattern*. If the input tree matches the pattern, the result is the replacement, otherwise the result is the (unchanged) input tree. For example in Figure 11, the `VBScript_ifthenelse_to_javascript` rule would only match and transform a VBScript `If` statement. To simplify the specification of the rules, `TXL` permits the marking of sub-trees in the pattern so they can be repeated in the replacement tree unchanged. In Figure 11 `EIB`, `EB`, `EXP`, and `STMTS` are examples of such markings.

Each transformation rule is localized to the matched statement and it would not affect other statements. The transformation rules are applied repeatedly on the parse tree until no more rules can be applied.

Figure 12 shows an example of the matching of the `VBScript_ifthenelse_to_javascript` rule to the parse tree generated for the example input file. The ovals in Figure 12 indicate the matching sub-trees for the nonterminals in the rule specified in Figure 11.

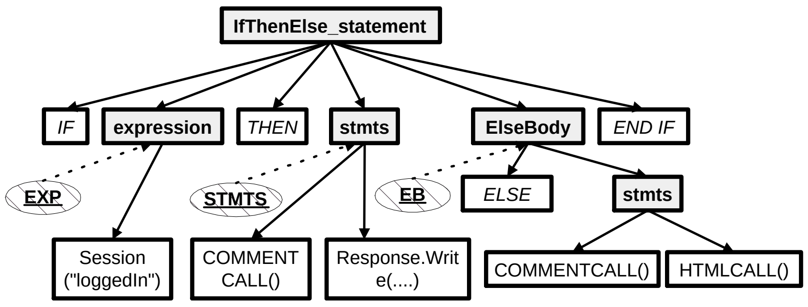

**Figure 12:** Matching Rule in Figure 11 to the parse tree shown in Figure 9

## 6 RELATED WORK

Migrating web applications between different technologies has been mainly done as a manual process. [19] and [31] present approaches to migrate web application which depend on Enterprise Java Beans (EJB) between application servers built by different companies. [3] describes a case study where a web application developed using windows technologies is migrated to a web application using java technology. Such approach are not automated, are error prone and are cumbersome for developers to perform, since they need to deal with many details that would ideally be automated.

Tools have been proposed to automate or semi-automate the migration [9, 23] of the database access code in a web application. Our approach is not capable of processing complex variations in the database access code between the source and target framework. Nevertheless, we can migrate database access code that is done using built-in framework objects. Thus these approaches complement our work, in particular for applications that use complex database access techniques such as IBM's Net.Commerce.

Since the publication of a preliminary version of this work in [17], a number of commercial approaches have been proposed to automate the migration of web applications. We briefly contrast these approaches to our work.

To deploy web applications written using Microsoft technologies on Unix platforms, Stryon offers a set of tools which reverse engineer the binaries (ILs) generated by Microsoft's .NET web framework into Java source code [18]. This Java source code along with a set of libraries provided by Stryon enable developers to run .NET web applications using Java technologies. This java source code produced by the application is not easy to maintain, as it is produced from the binary and is missing the documentation and code indentation. The java code may be missing some of the original code as it may have been removed during the optimization phase when the IL code was generated. The Stryon approach assumes that the developer continues using .NET languages to develop the application. The tool is only used to perform the migration for deploying the application on a Unix platform.

Other approaches similar to our approach of migrating the source code of the application have been proposed by Microsoft and by Diamond Edge. Microsoft introduced a migration guide to automate the migration of JSP web applications to the ASP.Net web applications [27]. The guide aims at facilitating the adoption of Microsoft's newly introduced web application framework (ASP.Net) by web developers. The guide defines the migration process into two phases: a functional equivalence migration phase and an optimization phase. The functional equivalence phase is largely automated using a tool which first performs a code analysis to identify any trouble spots in the migration process, then the tool automatically migrates the application. Unfortunately, no details are provided concerning the internals of the tool and the maintenance of comments in the migrated source code. Similarly to our approach, the tool attempts to migrate as much as possible of the application. Failing to do so it inserts tags in the source code for the migration engineer to review the troublesome areas. Once the initial phase is accomplished, the framework focuses on the manual optimization of the newly migrate site to benefit from the features of the ASP.Net platform.

Diamond Edge offers a set of tools (ASP2JSP) to migrate ASP web applications to JSP [2]. The approach followed by Diamond Edge is closely related to our approach. The ASP2JSP tool offers a mapping file to permit the mapping of user defined objects. Comments are not migrated between both applications.

## 7 OBSERVATIONS AND CONCLUSION

In this paper, we highlighted the risks faced by developers maintaining their web applications as they are locked into their initial development framework. These initial development framework may no longer be supported by the companies who produced them or may no longer suit the current requirements and future needs of a web application.

We proposed a migration approach to permit developers to migrate their web application to their desired modern framework. The approach has been used successfully to migrate the Hopper News application to NSP. The Hopper News application is a sample web application provided by Microsoft to showcase the ASP framework. It is described in more detail in [15].

Whereas prior migration research focused on migrating regular applications, the migration process of web applications faces additional challenges, in particular, the intermixing of programming languages inside web applications. Instead of proposing a new paradigm for migrating web applications, we focused on reusing prior migration techniques and results. To adopt prior migration techniques, we had to transform web applications from multilingual application to single language applications. We developed a process called *Water Transformations* to perform the transformation to single language applications. We then used traditional migration tools, such as `TXL`, for the migration. The approach automates error prone and low level migration details. Developers can concentrate on more interesting and complex problems in the migration process. The migration technique aims at easing future maintenance of the migrated code by preserving the structure of the migrated code and the relative position of comments in the generated code.

The *Water Transformations* approach can be used more generally to migrate programming languages that are embedded inside other programming languages (for example to migrate embedded SQL to newer version of SQL) by using the ideas of preprocessing the input file and placing place-holders to be later replaced in the *Postprocessing* stage after the transformations are done.

**ACKNOWLEDGEMENTS**
Web applications provided by Microsoft Inc. and Sun Microsystems of Canada Inc. were used to develop the grammar for the VBScript language, as no extensive reference manual describes the language in detail. In particular, we would like to thank Wai-Ming Wong from Sun for his assistance in our study of the web applications. We would also like to thank Thomas R. Dean, James R. Cordy from Queen's University and Andrew J. Malton from the University of Waterloo for introducing the TXL language to us.

## REFERENCES

[1] Macromedia, 2002. Available online at: `http://www.macromedia.com/software/coldfusion/`.

[2] ASP2JSP, The ASP to JSP Converter. Available online at: `http://www.diamondedge.com/products/Convert-ASP-to-JSP.html`.

[3] Best Practices - Case Studies: BlueNile.com – Building a Gem of a Site with Oracle and Sun. Available online at: `http://www.sun.com/third-party/`

global/oracle/success/BlueNileCaseStudy.pdf.

[4] A. Cimitile, U. de Carlini, and A. D. Lucia. Incremental migration strategies: Data flow analysis for wrapping. In *5th Working Conference on Reverse Engineering (WCRE)*, pages 59–68, Honolulu, Hawai, Oct. 1998.

[5] E-business implementation by industy sector, 1999. Available online at: http://ebusiness.mit.edu/cgi-bin/stats/catagorybrowser.cgi.

[6] J. Conallen. *Building Web Applications with UML*. object technology. Addison-Wesley Longman, Reading, Massachusetts, USA, first edition, Dec. 1999.

[7] J. Cordy, T. D. andA.J. Malton, and K. Schneider. Software Engineering by Source Transformation - Experience with TXL. In *IEEE 1st International Workshop on Source Code Analysis and Manipulation*, pages 168–178, Florence, Italy, Nov. 2001.

[8] J. Cordy, T. Dean, A. Malton, and K. Schneider. Software Engineering by Source Transformation. *Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology*, Feb. 2002.

[9] O. Corp. *In2j : Automated tool for migrating Oracle PL/SQL into Java*. 2001. Available online at: http://www.in2j.com.

[10] T. Dean, J. Cordy, K. Schneider, and A. Malton. Experience Using Design Recovery Techniques to Transform Legacy Systems. In *IEEE International Conference on Software Maintenance*, pages 622–631, Florence, Italy, Nov. 2001.

[11] ECMA - Standardizing Information and Communication Systems. Available online at: http://www.ecma.ch.

[12] Standard ECMA-262: ECMAScript Language Specification . Available online at: ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf.

[13] P. M. G. Mecca, P. Atzeni, and V. Crescenzi. The Araneus Guide to Web-Site Development - Araneus Project Working Report. AWR-1-99, University of Roma Tre, Mar. 1999. Available online at: http://www.dia.uniroma3.it/Araneus/publications/AWR-1-99.ps.

[14] A. E. Hassan. Architecture Recovery of Web Applications. Master's thesis, University of Waterloo, 2001. Available online at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/msthesis.pdf.

[15] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[16] A. E. Hassan and R. C. Holt. A Visual Architectural Approach to Maintaining Web Applications. *Annals of Software Engineering- Special Volume on Software Visualization*, 16, 2003.

[17] A. E. Hassan and R. C. Holt. Migrating Web Frameworks Using Water Transformations. In *Proceedings of COMPSAC 2003: International Computer Software and Application Conference*, Dallas, Texas, USA, Nov. 2003.

[18] iNET - Write Once in .NET, Run Anywhere. Available online at: http://www.halcyonsoft.com/products.asp?s=4.

[19] S. iPlanet. *Migration Guide, iPlanet Application Server*. 2000. Available online at: http://docs.sun.com/source/816-5774-10/mpreface.htm.

[20] Island Grammars, 2001. Available online at: http://losser.st-lab.cs.uu.nl/~visser/cgi-bin/twiki/view/Transform/IslandGrammars.

[21] I. Jacobson and F. Lindstrm. Re-engineering of old systems to an object-oriented archistecture. In *Object Oriented Programming Systems Languages and Applications Conference (OOPSLA)*, pages 340–350, Phoenix, Arizona, 1991.

[22] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: An experience report. In *IBM Centre for Advanced Studies Conference (CASCON)*, pages 1–13, Toronto, Canada, Nov. 1998.

[23] T. C. Lau, J. Lu, E. Hedges, and E. X. Xing. Migrating e-commerce database applications to an enterprise java environment. In *IBM Centre for Advanced Studies Conference (CASCON)*, Toronto, Canada, 2001. Available online at: http://www.cas.ibm.com/archives/2001/papers/cascon01/htm/english/abs/lau.htm.

[24] L. Moonen. Generating robust parsers using island grammars. In *Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001.

[25] L. Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Dec. 2002.

[26] L. Moonen. Lightweight impact analysis using island grammars. In *10th International Workshop on Program Comprehension*, Paris, France, 2002.

[27] JSP to ASP.NET Migration Guide, 2003. Available online at: `http://msdn.microsoft.com/asp.net/using/migrating/jspmig/`.

[28] P. Newcomb and P. Martens. Reengineering procedural into object-oriented systems. In *3rd Working Conference on Reverse Engineering (WCRE)*, Toronto, Canada, July 1995.

[29] Business Flow Accelerators. Available online at: `http://www.oracle.com/consulting/offerings/ebs/index.html?content.html`.

[30] P. Patil, Y. Zou, K. Kontogiannis, and J. Mylopoulos. Migration of Procedural Systems to Network-Centric Platforms. In *IBM Centre for Advanced Studies Conference (CASCON)*, pages 68–82, Toronto, Canada, Nov. 1999.

[31] T. Research. *Moving from IBM WebSphere 3 to BEA WebLogic Server 5.1, White Paper*. Available online at: `http://www.bea.com/products/weblogic/server/Migration_WP.pdf`.

[32] H. Sneed. Encapsulating legacy software for use in client/server systems. In *3rd Working Conference on Reverse Engineering WCRE)*, pages 104–119, Nov. 1996.

[33] H. Sneed. Object-oriented cobol recycling. In *3rd Working Conference on Reverse Engineering (WCRE)*, pages 169–178, Nov. 1996.

[34] The TXL Transformation System, 2001. Available online at: `http://www.txl.ca/`.

[35] A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from conventional procedural language. In *2nd Working Conference on Reverse Engineering (WCRE)*, pages 227–236, Toronto, Canada, July 1995.