A Cross-Language Framework for Developing AJAX Applications

Arno Puder
San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
arno@sfsu.edu

ABSTRACT

AJAX (Asynchronous JavaScript And XML) applications have received wide-spread attention as a new way to develop highly interactive web applications. Breaking with the complete-page-reload paradigm of traditional web applications, AJAX applications rival desktop applications in their look-and-feel. AJAX places a high burden on a web developer requiring extensive JavaScript knowledge as well as other advanced client-side technologies. In this paper, we introduce a technique that allows a developer to implement an application in Java or any .NET language and then automatically cross-compile it to an AJAX-enabled web application.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers

General Terms

Languages

Keywords

Web Applications, AJAX, Cross-Compiler

1. INTRODUCTION

The initial intent of the World-Wide Web (WWW) was to provide access to remote documents. The HTML standard that is used to describe content for the WWW was quickly extended to include user interface elements such as buttons and input fields that allowed the construction of web applications where the web browser acts as a generic client. Although enhanced by user interface elements, the look-and-feel of web applications still was far from the user experience of a regular desktop application.

With the release of Google Maps, a new era of web applications began. Instead of viewing a web application as a sequence of mostly static HTML pages that are loaded in response to user interaction, the latest generation of web applications made extensive use of JavaScript inside a browser to create highly interactive applications that rival desktop applications with their look-and-feel. In fact, some new web applications using this paradigm implement applications such as word processors or spreadsheets that traditionally have only been available on the desktop. The browser as a generic client slowly becomes the next desktop. The acronym AJAX (Asynchronous JavaScript And XML, see [7]) is used to describe these new generation web applications.

Writing AJAX applications requires extensive knowledge of JavaScript, DOM manipulations, and portability issues across different browsers. The lack of IDEs and appropriate skill-set among current IT developers make the development of AJAX applications a daunting task. The basic assumption of this paper is that JavaScript is the assembler language of the web. Ideally one does not want to be exposed to it. In this paper, we introduce XML11 that features a cross-compiler capable of translating regular Java applications to AJAX applications. The outline of this paper is as follows: Section 2 gives a detailed description of the XML11 framework describing its various components. Section 3 gives an overview of the prototype implementation of XML11. Section 4 discusses related work and in Section 5 we provide a conclusion and outlook.

2. XML11 FRAMEWORK

The name of our framework is XML11 and it is inspired by the old X11-Windows protocol developed by MIT in 1984. Just like an X11-Server can render any user interface, a web browser likewise serves as a generic client that can render arbitrary user interfaces. Whereas the X11-Protocol focuses on graphics (i.e., the X11-Protocol has no notion of buttons or listboxes) the XML11-Protocol supports widgets. The client-side components of XML11 are implemented in JavaScript and are loaded after visiting a respective URL pointing to the XML11 Server. Unlike in the X-Windows world, a web browser contains a Turing complete execution platform through a JavaScript interpreter. Note that we do not consider Java or Flash extensions because the idea of AJAX is to only use what is commonly available in all browsers. For this reason the client-side portions of XML11 are implemented in JavaScript and XML11 cross-compiles those parts of the application to JavaScript that are to be migrated to the browser. The complete XML11 framework is explained

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

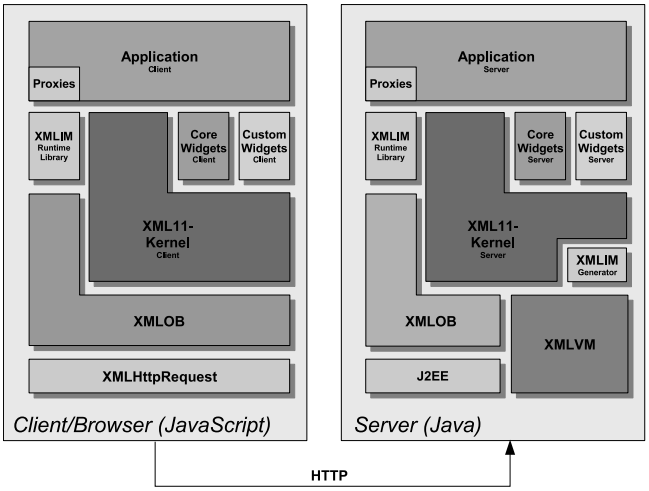

Figure 1: Architecture of XML11.

in detail in the following sections.

2.1 Architecture Overview

Figure 1 depicts the overall architecture of XML11. It follows the Client/Server model where the client is the browser and the server is implemented as a J2EE Application Server. The client uses HTTP requests to interact with the remote server. Upon visiting an XML11-enabled application, the complete client-side JavaScript implementation of XML11 is first downloaded into the browser. Also those parts of the application that should be migrated to the client, are automatically cross-compiled to JavaScript as explained later.

As can be seen in Figure 1, the client and server are largely symmetrical in their internal structure. The client is completely implemented in JavaScript, whereas the server-side of XML11 is implemented in Java. In the following we give a brief overview of the various layers making up the XML11 architecture beginning with the lowest layer. Some of those layers will be discussed in detail in subsequent sections.

The bottom layer shown in Figure 1 implements the transport mechanism for XML11. Since XML11 runs inside a browser, the transport mechanism uses the `XMLHttpRequest` object on the client-side to issue HTTP requests to the server. The server is implemented as a J2EE Application Server, accepting incoming HTTP requests from the client. The layer above the transport mechanism is XMLOB, an XML-based Object Broker. The purpose of XMLOB is to create an abstraction from the raw transport mechanism by offering a simple message-based, bi-directional communication model between objects. XMLOB is capable of sending messages asynchronously in both directions. Section 2.2 will explain XMLOB in detail and in particular how the server can send messages asynchronously to the client.

One component that only exists on the server-side is the Java-to-JavaScript cross-compiler. This component is called XMLVM because it is based on an XML-based Virtual Machine model. The task of XMLVM is to translate the client-side portions of an application written in Java or C# to JavaScript. The cross-compiled application is then migrated to the client. The application shown in Figure 1 on the client-side is the result of this translation process. XMLVM is explained in detail in Section 2.3.

The core part of XML11 is based on a micro-kernel architecture. The idea is to create an architecture where new functionality can be plugged into the system dynamically at runtime. The XML11-core is mainly responsible for the plugin management. Various plugins enrich XML11 with additional functionality. One plugin implements the core widgets that an application can use for its user interface. In the current prototype implementation of XML11 those core widgets have the same API as the Abstract Windowing Toolkit (AWT). As a consequence, any AWT application that was originally developed for the desktop can run under XML11.

Some plugins implement custom widgets (such as a Google Map panel) that serve as the foundation of mashups in XML11. Other plugins are created dynamically. One example are the application-specific proxies that support the transparent distribution of the client- and server-side of the application. The idea of dynamically creating the proxies at runtime leads to our notion of an implicit middleware (XMLIM) that is further explained in Section 2.4.

2.2 Object Broker (XMLOB)

XMLOB offers middleware services to the applications running on top of it. It is not meant to be a full-fledged middleware such as CORBA or Web Services, but rather intended as a light-weight solution sufficient for AJAX applications. Figure 2 depicts the basic architecture of XMLOB. Objects register with an Object Broker that is responsible for message transmission and dispatching. Objects on the side of the client are implemented in JavaScript whereas the objects on the server-side are implemented in Java. Objects are addressed by *a priori* known unique identifiers to avoid the need of a naming service.

One distinguishing feature of XMLOB is that its messaging service is completely symmetrical: objects on either side can send any other object a message. This is obvious for JavaScript objects calling Java objects; the Object Broker inside the browser can make use of the `XMLHttpRequest` object to send a message to the server. The question is how an object on the server-side can send a message to an object on the client-side. The problem is that HTTP is a client/server protocol and only the client can initiate HTTP requests.

The solution to this problem is a technique called deferred-reply: the Object Broker on the client-side issues a HTTP request, but upon receiving this request on the server-side, the Object Broker simply defers the reply until there is a message to be sent back to the client. This guarantees immediate transmission of messages from the server to the client. The downside of this approach is that at any point in time there is an open HTTP request in order to guarantee that messages can immediately be sent to the client. This solution does not scale for large number of clients when each of the clients is using this deferred reply technique.

Deferred replies allow asynchronous updates that can be pushed from the server to the client. Many applications do not need this feature and therefore it is not necessary for using deferred replies. In order to accommodate different application needs, XMLOB supports three different communication models:

- **Asynchronous:** This model employs the aforementioned deferred-reply technique. At any point in time, XMLOB keeps one HTTP connection open so that

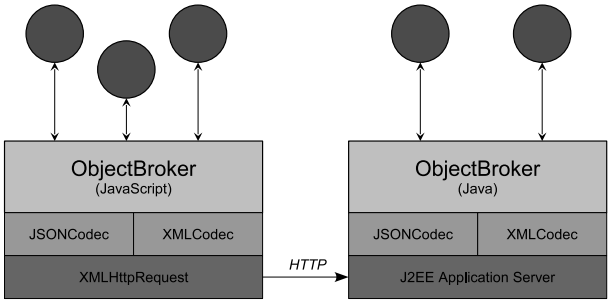

Figure 2: Architecture of XMLOB.

messages originating on the server-side can immediately be transmitted to the client.

- **Synchronous:** Using this model, XMLOB will only issue HTTP requests when a message is to be sent to the server. Any messages pending on the server-side will be piggy-backed onto the HTTP response. While this model scales to a large number of clients, messages originating at the server will be queued until the client issues a HTTP request.

- **Polling:** The polling model is a hybrid between the aforementioned asynchronous and synchronous communication models. Using the polling policy, the XMLOB client will periodically poll for messages pending for transmission at the server. This is the best compromise between interactivity of applications and scalability of the communication infrastructure.

Every application using XMLOB can choose between one of those three communication models that best suits its needs. A highly interactive application with asynchronous updates will use the asynchronous communication model. An application where the user interface only updates in response to interaction from the end-user will use the synchronous communication model.

The code fragment below shows how to send a message from the server (written in Java) to the client (written in JavaScript). XMLOB offers an appropriate API to make construction and handling of messages as efficient as possible.

```
1 // Java - Sending a message
2 Message msg = new Message("awtManager",
3                           "createWidget");
4 msg.put("id", "ELEM_1");
5 msg.put("type", "button");
6 msg.send();
7
8 // JavaScript - Receiving a message
9 function AWTManagerClass {
10   this.createWidget = function(msg)
11   {
12      var id   = msg.id;
13      var type = msg.type;
14      // ...
15   }
16 }
17 XMLOB.registerObject("awtManager",
18                       new AWTManagerClass());
```

The Java code in lines 2–6 construct and send a message. Every message has a target object identifier (`awtManager`) and method that is supposed to be called on the target object (`createWidget`). Those parameters are specified in the constructor of class `Message`. Actual parameters can be added in form of name/value pairs. In the example above, parameters `id` and `type` are added to the message before the message is being sent to the client. Depending on the communication model chosen by the application, the message will eventually be transported to the client-side. The Object Broker on the client-slide will unmarshal the message and invoke the appropriate method on the target object. Lines 9–18 in the code fragment above show the JavaScript code for instantiating and registering the new object with XMLOB. Note that the actual parameters can be referenced as properties of a JavaScript object. This provides a natural mapping and makes best use of JavaScript as an interpreted language.

The above example only shows very simple actual parameters. The data model supported by XMLOB is inspired by JSON (see [3]). There are two different types of parameters:

Objects: unordered name/value pairs.

Arrays: list of values.

Both data types can be arbitrarily nested. While this data model does not provide the same flexibility as a full-fledged middleware such as CORBA, in our experience this sufficient for AJAX applications. XMLOB supports two different marshalling engines as shown in Figure 2. The JSON codec marshalls the actual parameters as a JavaScript data type and the XML codec marshalls the actual parameters as an XML document. The fragment below shows the resulting PDU that would be observed on the wire based on the simple example discussed earlier where a message is sent to an object whose identifier is `awtManager`:

```
1 // JSONCodec
2 {xmlob:{
3     message:[
4       {method: "createWidget",
5        target: "awtManager",
6        id    : "ELEM_1",
7        type  : "button"}]}}
8
9 <!-- XMLCodec -->
10 <ob:xmlob xmlns:ob="http://www.xml11.org/xmlob/">
11   <ob:message ob:target="awtManager"
12               ob:method="createWidget"
13               id="ELEM_1" type="button"/>
14 </ob:xmlob>
```

2.3 Cross-Compiler (XMLVM)

A key component of the XML11 architecture is the code migration framework that allows to migrate application logic into the browser. This is done by cross-compiling the client-side portions of the application to JavaScript and then migrating the generated code to the browser. We are able to cross-compile both Java class files as well as .NET executables to JavaScript. The translation process happens in two stages. The binary executable (either a Java class file or a .NET executable) is first translated to an XML-based programming language that is modeled after a stack-based machine common to both the Java and .NET virtual machines.

We call this language XMLVM. Once the XMLVM program is generated, it can be mapped to other languages such as JavaScript via stylesheets. These two stages are explained in detail in the following two sections.

2.3.1 XML Representations of Byte Code Instructions

The foundation of the code migration framework within XML11 is an XML-based programming language. Since the semantics of this language is modeled after a virtual machine (see [11]), we call it XMLVM. XMLVM basically allows us to represent either the contents of a Java class file (see [11]) or the contents of a .NET executable (see [6]) through XML. Another way to look at XMLVM is that it defines an assembly language for these virtual machines. Although both platforms share many things in common, their byte code differs nonetheless. XMLVM uses XML-namespaces to clearly distinguish between byte code instructions stemming from either platform. The following template shows the general structure of an XMLVM translation unit:

```
<xmlvm xmlns:jvm="http://xml11.org/jvm"
       xmlns:clr="http://xml11.org/clr">
  <class ...>
    <field .../>
    <method ...>
      <signature>...</signature>
      <code>...</code>
    </method>
  </class>
</xmlvm>
```

An XMLVM program consists of several classes, each contained in a separate translation unit. Each class can have one or more fields and methods. The attributes of the XML-tags, that are not shown in the template above, give more details such as identifiers or modifiers. A method is defined through a signature and the actual implementation, denoted by the tags <signature> and <code> respectively. Consider the following simple Java-class whose only static method determines if an integer is odd (see [1]):

```
// Java
public class Check
{
    static public boolean isOdd(int i)
    {
        return i % 2 != 0;
    }
}
```

Class `Check` has one static public method called `isOdd`. The method returns a boolean value indicating whether the actual integer parameter is odd or not. Although this is a very simple example, it allows us to show all basic aspects of an XMLVM program. The following simplified XML shows the representation of class `Check` in XMLVM:

```
<xmlvm xmlns:jvm="http://xml11.org/jvm">
  <class name="Check">
    <method name="isOdd" stack="2" locals="1">
      <signature>
        <return type="boolean" />
        <parameter type="int" />
      </signature>
      <code>
        <jvm:iload type="int" index="0" />
        <jvm:iconst type="int" value="2" />
        <jvm:irem />
        <jvm:ifeq label="0" />
        <jvm:iconst type="int" value="1" />
        <jvm:goto label="1" />
        <jvm:label id="0" />
        <jvm:iconst type="int" value="0" />
        <jvm:label id="1" />
        <jvm:ireturn />
      </code>
    </method>
  </class>
</xmlvm>
```

It should be emphasized again that the above XMLVM program is essentially an XML-representation of the contents of the `Check.class` class file generated by the Java compiler. The top-level tags are identical to the XML-template shown earlier. The <`method`>-tag has two attributes: `stack` and `locals`. `stack` tells the virtual machine the maximum stack-size needed for this method. In this example, method `isOdd` will never push more than 2 elements onto its stack. The `locals` attribute tells the virtual machine how many local variables are needed for this method. Actual parameters are automatically copied by the virtual machine to local variables upon invoking the method. Since there is only one input parameter, only one local variable is needed. Note that the Java compiler computes the values for `stack` and `locals` and stores them in the class file.

The more interesting part of the XMLVM-program shown above is the actual implementation of method `isOdd`. The byte code instructions are prefixed with XML-namespace `jvm:` to indicate that they belong to the Java virtual machine. .NET byte code instructions are prefixed with `clr:` (for Common Language Runtime). The <`jvm:iload`> (*integer load*) instruction pushes the actual parameter of the method, referred to by local variable with index 0, onto the stack. Instruction <`jvm:iconst`> (*integer constant*) pushes a constant referred to by attribute `value` onto the stack. The next instruction <`jvm:irem`> (*integer remainder*) pops off the last two values (the actual parameter and the constant 2) and pushes their remainder after division back onto the stack. The <`jvm:ifeq`> (*if equal*) instruction pops the last element off the stack and performs a conditional jump if its value is equal to 0. Note that flow control is represented in XMLVM through gotos. The <`jvm:ireturn`> (*integer return*) instruction pops off the top of the stack and returns the value to the caller of method `isOdd`.

It should be noted that the XMLVM instruction set features a mix of low-level and high-level virtual machine instructions. In addition to the low-level instructions mentioned above, there exist high-level instructions such as <`jvm:new`> (for instantiating new objects) and <`jvm:invokevirtual`> (invoke a virtual method). These instructions go beyond the capabilities of normal (hardware) machine languages and therefore require substantial runtime support. Microsoft's .NET platform features a similar mix of byte code instructions for its virtual machine, however there are subtle differences. E.g., whereas the Java virtual machine features typed instructions for numerical operations (e.g., <`jvm:irem`> for *integer remainder*, <`jvm:frem`> for *float remainder*, and <`jvm:drem`> for *double remainder*), the .NET platform only has one generic untyped <`clr:rem`> instruction. The precise type of the operands has to be determined through a

data flow analysis.

2.3.2 Code generation

As stated earlier, XMLVM can be seen as an assembly language for the Java virtual machine. The difficult part is done by a Java compiler. Once a class file has been created as the result of the compilation process, it can be easily translated to XMLVM simply by analyzing the contents of the class file. The next step consists in translating XMLVM to another programming language such as JavaScript or ActionScript. This translation can be done by an XSL-stylesheet that maps XMLVM-instructions one-to-one to the target language. Since XMLVM is based on a simple stack-based machine, we simply mimic a stack-machine in the target language. An example helps to illustrate this approach. The XMLVM instruction <jvm:irem> introduced earlier pops off two values and pushes the remainder after division back onto the stack. Here is the XSL-template that creates JavaScript code for this instruction:

```
<xsl:template match="jvm:irem">
        <xsl:text>
            __op1 = __stack[--__sp];
            __op2 = __stack[--__sp];
            __stack[__sp++] = __op2 % __op1;
        </xsl:text>
</xsl:template>
```

We mimic the virtual machine of XMLVM via the variables `__locals` (for local variables), `__stack` (for the stack), and `__sp` (for the stack pointer). Variables `__op1` and `__op2` are used as temporary variables needed by some XMLVM-instructions. Those variables are declared for every method. Using stylesheets to translate XMLVM instructions to the target language works as long as there is a corresponding instruction. In some instances this translation has to be done carefully in order to retain the semantics of the original instruction. E.g., XMLVM features an instruction for adding two integers, <jvm:iadd>, that works analogous to <jvm:irem>. However, it is not correct to map addition to the +-operator in JavaScript. The problem is that XMLVM (since it is based on the Java VM) treats integers as 4-byte values, whereas JavaScript has only one numeric type and does not distinguish between integers and floating-point values. This difference has to be accounted for by using a wrapper function written in JavaScript that retains the original semantics of <jvm:iadd>.

Another interesting problem is that flow control in XMLVM is based on jump instructions, whereas JavaScript does not feature a goto-statement. There is no straightforward way to map the XMLVM <jvm:ifeq> instruction discussed earlier to JavaScript via a stylesheet. To solve this problem, XMLVM offers a transformation tool that removes goto-statements from a program and replaces them with loop-, break-, and continue-instructions. The goto-elimination algorithm is based on an old paper by Ramshaw (see [14]). Using Ramshaw's algorithm, XMLVM first eliminates all gotos. Using our previous example, here is the result of this transformation:

```
<code>
  <jvm:iload type="int" index="0" />
  <jvm:iconst type="int" value="2" />
  <jvm:irem />
  <fc:loop id="0">
    <fc:loop id="1">
      <fc:break condition="ifeq" id="1" />
      <jvm:iconst type="int" value="1" />
      <fc:break id="0" />
    </fc:loop>
    <jvm:iconst type="int" value="0" />
    <fc:break id="0" />
  </fc:loop>
  <jvm:ireturn />
</code>
```

As can be seen, the goto-elimination algorithm introduces new XML-tags for flow control. Since these tags do not belong to the original XMLVM instruction set, they are placed in their own XML-namespace. Note that in the example above, the <jvm:ifeq> from the original program has been replaced by two nested loops. Once gotos have been removed, it is relatively straight forward to map flow control statements to JavaScript instructions via stylesheets. The code below represents the JavaScript version of the class `Check` after applying all necessary stylesheets:

```
// JavaScript generated by stylesheet
function Check()
{
    Check.isOdd = function(__arg1)
    {
        var __locals = new Array(1);
        var __stack = new Array(2);
        var __sp = 0;
        var __op1;
        var __op2;
        __locals[0] = __arg1;
        __stack[__sp++] = __locals[0];
        __stack[__sp++] = 2;
        __op1 = __stack[--__sp];
        __op2 = __stack[--__sp];
        __stack[__sp++] = __op2 % __op1;
        label0: while (1) {
            label1: while (1) {
                __op1 = __stack[--__sp];
                if (__op1 == 0) break label1;
                __stack[__sp++] = 1;
                break label0;
            }
            __stack[__sp++] = 0;
            break label0;
        }
        return __stack[--__sp];
    }
}
```

The JavaScript code above was generated automatically by applying an appropriate XSL-stylesheet to the XMLVM version of class `Check`. As can be seen, there is a natural mapping from XMLVM to JavaScript. The intention is not to generate readable code, but correct code that uses the semantics of the target language. It should also be obvious that the above JavaScript code will be less efficient than the original Java program. Our assumption is that we do not migrate computational heavy applications to the browser.

Figure 3 shows the complete XMLVM toolchain. The XMLVM featuring Java byte code instructions is labeled XMLVM_{JVM} whereas XMLVM_{CLR} uses byte code instructions from the .NET platform. Programs for the CLR can be converted to the Java platform by doing a data flow analysis as described earlier. Once a XMLVM_{JVM} program has

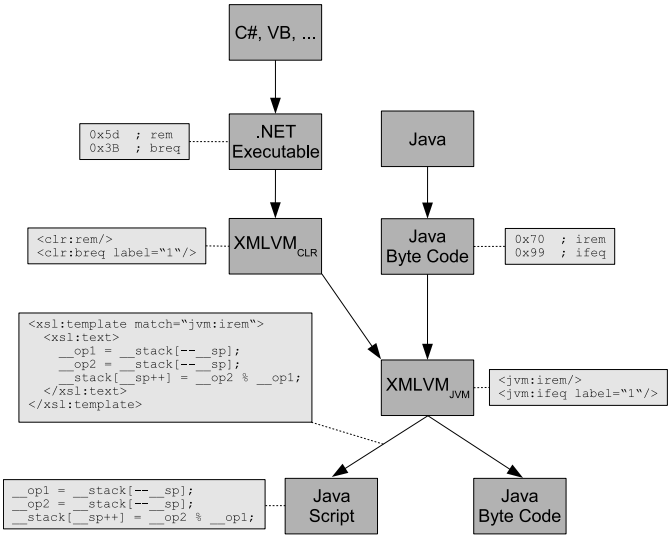

Figure 3: XMLVM code translation toolchain.

been created, it can be mapped to JavaScript as described above, or it can be mapped back to a Java class file. The latter is useful when doing further transformations on the XMLVM_{JVM} program like the one described in the following section.

2.4 Implicit Middleware (XMLIM)

The code migration framework introduced in the previous section cannot generally be used to migrate a complete application. The reason for that is that most applications are not self-contained but require access to fixed resources that cannot be migrated. One example are databases that need to reside on the server. An application can therefore be partitioned into classes that can be migrated to the client and classes that need to remain on the server-side. The decision which classes to migrate is currently determined by a configuration file that the application programmer has to provide.

Problems arise when a class that has been migrated to the client is referencing a class whose implementation resides on the server. What is needed in this case is a middleware that marshalls the actual parameters provided by the client and sends them to the server. This task is typically done by a proxy that is linked with the application and that appears to the application as if it were the remote object. The proxy can use XMLOB to transport the marshalled actual parameters.

The problem still remains how this proxy is generated. The proxy has the same API as the remote object and is therefore application specific. I.e., depending on the signatures of the methods offered by the remote object at its public interface, a specific proxy has to be generated. Traditionally, the generation of the proxy is dependent on some external artifact describing the public interface of an object. E.g., CORBA has the Interface Definition Language (IDL) for that purpose and Web Services use the Web Services Definition Language (WSDL) to achieve the same. A tool generates the proxies based on the IDL- or WSDL-specifications.

As seen in the section explaining XMLVM, detailed information on the signatures is already present in an XMLVM program. E.g., in the example discussed in Section 2.3.1 it can be seen that the signature of method `isOdd()` is contained in the XMLVM representation of class `Check` (denoted by XML-tag <`signature`>). Using the signature information present in an XMLVM program, it is possible to automatically generate the appropriate proxy. The reason we call this approach implicit is because the proxy can be generated automatically without requiring an external specification such as IDL or WSDL. From an application programmers perspective the middleware is implicitly inserted to connect remote objects.

Proxies have to be generated in the same language as the application that uses them, because they need to be linked to the application. Since XML11 requires JavaScript on the client-side and Java on the server-side, proxies also need to be generated for these two languages. The way this is done is by expressing the implementation of the proxy through XMLVM_{JVM} and then use the regular stylesheets to generate the proxy for the target language. Based on the signature of a method, the implementation between the <`code`> tags is replaced with one that marshalls the actual parameters and that sends it to the remote object via XMLOB. This implementation can make use of any XMLVM_{JVM} byte code instruction so that it can later be mapped via the stylesheet to the target language.

3. PROTOTYPE IMPLEMENTATION

The prototype implementation of XML11 features a plugin that is API-compatible with Sun Microsystem's Abstract Windowing Toolkit (AWT). As a consequence, any AWT application can run under XML11 on the server-side. If the application instantiates a button (class `java.awt.Button`), the plugin will intercept this call and create an appropriate message to the client-side plugin implementing the AWT referred to as the `awtManager` in an earlier example. We have implemented wrapper classes that expose functionality of the Microsoft WinForm GUI through Java API. This allows a .NET program using WinForms to be cross-compiled to Java byte code via XMLVM_{CLR} that is then transformed to XMLVM_{JVM}.

The client-side, implemented in JavaScript, uses DOM manipulations to create GUI elements with the specified parameters. We make use of the Qooxdoo library for those DOM manipulations (see [13]). When the user interacts with the application inside the browser and the appropriate application logic has not been migrated to the client, a message describing the event will be sent back to the server. The XML11 server will then feed the event into the application's event queue where it will finally be processed.

The implementation of XMLVM is leveraging two Open Source libraries that allow to parse Java class files as well as .NET executable. For Java class files we use the BCEL (Byte Code Engineering Library, see [4]) and for .NET executables we use MBEL (Microsoft Byte code Engineering Library, see [15]). We also use BCEL to generate Java class files when cross-compiling from .NET executables to the Java virtual machine. The data flow analysis and byte code transformation required for this are also done using XSL stylesheets.

We have also implemented a non-trivial application to demonstrate a code migration scenario. This application lists movies made in San Francisco (see [9]). For each movie, details such as release year and synopsis are shown. Additionally, locations where certain scenes of those movies

were shot, are displayed as markers in a map of San Francisco. This map is an example of a custom widget plugin for XML11. We make use of Google Maps for this task.

The SF-Movies application is completely written in Java using the AWT. Proxy classes offer a Java-API to Google Maps during development of the application. When the application is cross-compiled to JavaScript, those proxies are replaced with the actual Google Maps implementation. The AWT portion of SF-Movies consists of nine Java classes (not counting the aforementioned Google Maps proxies) with a total of 1684 lines of Java code. The resulting XMLVM of this Java code is 8649 lines of XML. This XML is then cross-compiled via an appropriate stylesheet to 12008 lines of JavaScript code that will be downloaded to the browser. This results in an increase in code size of factor 7 when cross-compiling from Java to JavaScript code for this particular application. We have observed similar increase in code size for other applications.

While this results in a significant increase, we have observed that network latencies play a more significant role for web applications. The 12000 lines of code can be compressed to 22 kB which guarantees fast download times. Interactive applications will not suffer from the added overhead of simulating a stack machine. But it would not be feasible to cross-compile computationally intensive applications. But it is questionable if those kind of applications should be cross-compiled to run inside a browser in the first place. In order to reduce the size of the generated code, we are considering to convert the stack-based XMLVM code to a register-based machine. This will be done as future work.

4. RELATED WORK

Several projects – commercial and Open Source – exist that aim at providing an easy migration path for legacy Java applications to web applications. WebCream is a commercial product by a company called CreamTec (see [2]). They have specialized in providing AWT and Swing replacements that render the interface of the Java application inside of a web browser. WebCream makes use of proprietary features of Microsoft's Internet Explorer and therefore only runs inside this browser.

Several Open Source projects follow the same idea of exposing Java desktop applications as web applications. One project is called WebOnSwing (see [12]). Unlike WebCream, this project is not tailored for a particular browser. One feature offered by WebOnSwing are templates that allow to change the look-and-feel of the application that is rendered inside the browser. Another project with similar features, but not quite as mature, is SwingWeb (see [10]). A third project called RAP (Rich AJAX Platform, see [5]) is part of the Eclipse framework. Similar to WebOnSwing and SwingWeb, RAP builds upon the SWT API. Processing happens on the server-side taking advantage of the OSGi framework.

The major difference between these approaches and the one introduced in this paper is that none of them supports code migration. While the user interface rendered inside the browser is similar to that of a desktop application, every event such as pushing a button, requires an HTTP request to the remote server. Migrating the application logic to the browser dramatically increases the responsiveness of the application while reducing the load on the remote server.

Google has recently released the Google Web Toolkit (GWT, see [8]) that is also based on a cross-compilation approach.

	XML11	**GWT**
Philosophy	Desktop Applications	Web Applications
Cross-Compiler	Byte Code Cross Compiler	Source Code Cross Compiler
Linking	Class loader	Static linking
Widget Toolkit	AWT Replacement	Proprietary Widgets
Debugging	Native Debugger	IE

Table 1: XML11 vs. GWT comparison.

GWT translates from Java to JavaScript, but there are several differences to XML11. GWT places great emphasis that the application behaves like a web application. It is possible to place bookmarks into the application. XML11 on the other hand follows the philosophy of desktop applications. Via XMLOB, it is possible to push updates to the browser asynchronously (which GWT cannot do) and therefore an XML11 application has more the look-and-feel of a desktop application.

Another difference between XML11 and GWT is the way the Java application is translated to JavaScript. GWT uses a source code level cross-compiler. This requires GWT to parse Java source code. Currently GWT only supports Java 1.4. Since XMLVM begins on byte code level, new features introduced in Java 5 (such as generics or annotations) are already supported (since generics are handled inside the JVM with existing byte code instructions). Beginning the translation process on the byte code level also makes other features such as the implicit middleware easier to implement, since XMLVM already offers all required information in easy to parse XML-markup.

XML11 features a class loader written in JavaScript that mimics the semantics of the JVM class loader. Only when a class is needed by an application, it will send a request to the remote XML11 server. GWT only allows static linking where the developer has to choose which classes belong to an application. All these classes are loaded during the first visit of the GWT application. Consequently GWT does not support reflection.

Another major difference between XML11 and GWT that also affects application programmers has to do with the widgets used to build a user interface. GWT relies on their own widget library. I.e., GWT introduces a Google-button, a Google-listbox, etc. XML11 on the other side leverages existing GUI libraries such as AWT or Swing. The benefit of our approach is that application developers can use their existing skill-set without having to learn the details of another GUI library. Our approach also makes it possible to debug the application using existing tools as if it were a desktop application. Table 1 summarizes the main differences between GWT and XML11.

5. CONCLUSIONS AND OUTLOOK

Building AJAX applications is a daunting task because of complexities of various technologies required to write end-to-end applications. In particular the JavaScript language itself as well as cross-browser portability issues make the development of AJAX applications difficult. A common solution in computer science is to create proper abstractions

that abstract away from the complexities of the underlying system. In that sense, we view JavaScript as the assembler language of the web. XML11 allows a developer to implement an AJAX application in Java or any of the .NET programming languages that is then cross-compiled to portable JavaScript.

The prototype implementation of XML11 replaces Sun Microsystems AWT. Future work will include other plugins to support Swing or SWT. More work is also planned on the cross-compiler. Since XMLVM is based on a stack machine model, the way the code generated during cross-compilations mimics a stack machine which creates unnecessary verbose JavaScript code. Using data-flow analysis it will be possible to transform the stack-based to a register-based machine which would greatly optimize the generated JavaScript code. The prototype implementation of XML11 including various sample applications are available under an Open Source license from `http://www.xml11.org/`.

6. REFERENCES

[1] Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley Professional, June 2005.

[2] CreamTec, LLC. *WebCream*. http://www.creamtec.com/webcream/.

[3] Douglas Crockford. *JSON - JavaScript Object Notation*. http://json.org/.

[4] Markus Dahm. Byte code engineering. Java Informations Tage, pages 267–277, 1999.

[5] Eclipse Foundation. *Rich AJAX Platform*, 2006. http://www.eclipse.org/rap/.

[6] ECMA. *Common Language Infrastructure (CLI)*, 4th edition, June 2006.

[7] Jesse Garrett. *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[8] Google. *Google Web Toolkit - Build AJAX apps in the Java language*. http://code.google.com/webtoolkit/.

[9] Sascha Häberling and Arno Puder. *Movies made in San Francisco*, 2007. http://www.sf-movies.org.

[10] Tiong Hiang Lee. *SwingWeb*. http://swingweb.sourceforge.net/swingweb/.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Pub Co, second edition, April 1999.

[12] Fernando Petrola. *WebOnSwing*. http://webonswing.sourceforge.net/xoops/.

[13] Qooxdoo. *Open Source AJAX Framework*, 2006. http://www.qooxdoo.org.

[14] Lyle Ramshaw. Eliminating goto's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.

[15] Michael Stepp. *MBEL: The Microsoft Bytecode Engineering Library*. http://www.cs.arizona.edu/mbel/.