XSL Transformations: A Delivery Medium for Executable Content Over the Internet

RUHSAN ONDER
Computer Engineering Department
Eastern Mediterranean University, Famagusta, North Cyprus

ZEKI BAYRAM
Computer Engineering Department/Internet Technologies Research Center
Eastern Mediterranean University, Famagusta, North Cyprus

{Ruhsan.onder, Zeki.bayram}@emu.edu.tr

Abstract. Sending code over the Internet to be executed on client Web browsers naturally requires a processor for the programming language of the code to be already present on the client browser. Code authors are thus restricted to widely-supported programming languages such as JavaScript, Java (as Applets) or VB-Script. We propose a scheme in which *any* language is utilized by the code author, as long as the processor for the code is packaged with it when the code is sent to the client for execution. As an instance of this scheme, we define an XML-based imperative language XIM, and implement its processor in XSLT. The possibilities offered by this approach are demonstrated in a WEB application in which the user interactively selects the kind of service needed from the server and provides the parameters for the service. The server, in turn, generates the code to be executed on the client. Possible uses of this scheme are specifying the semantics of Web services, implementing distributed computing, and specialized programming tasks which would be better implemented in some special-purpose programming language, which is not expected to be widely supported in Web browsers.

1 Introduction

When sending code to be executed in a remote computer, the code author needs to make assumptions regarding the availability of a certain hardware/software configuration and environment for the code to run properly. Platform independence- the ability of the code to run on any platform without modification- is thus a major issue. To address this issue, a commonsense approach is to provide an abstract machine for compiled code to run in, and make sure that all remote users who want to run the remotely delivered programs have the environment installed already. Examples of this approach is the JAVA [1] and the .NET [2] platforms. Installing these environments on client computers requires substantial resources, in terms of hardware configuration, disk usage etc., however.

If we limit our attention to executing code embedded in Web pages (i.e. *dynamic web pages* with client-side scripting), we have scripting languages that are already built into Web browsers. JavaScript [3] and VBScript [4] are two well-known scripting languages supported by widely-used browsers. However, these languages suffer compatibility problems, and a lack of standardization. A given script in JavaScript needs to be written in more than one version, if it is to run on different browsers without any problems. Yet others, such as VBScript, enjoy support only in a limited number of browsers. An alternative, widely used technology for executing code on the client side is Java Applets. This provides the programmer the capabilities of a full-scale programming language (i.e. Java). Still, Java applets require the Java Runtime Environment, which consumes substantial resources on the client side.

Browsers have another kind of language processor installed on them - that for Extensible Stylesheet Language Transformations XSLT [5]. XSLT stylesheets are applied to XML documents, mainly for the purpose of generating XHTML pages for presentation on browsers. Thus, XSLT helps to separate content (inside XML documents) from presentation (XHTML documents). As such, XSLT can hardly be seen in the same class as Scripting languages, such as JavaScript. However, XSLT has enough features to make it suitable for the implementation of the operational semantics of an imperative language.

XIM is an XML based programming language, with imperative control features such as assignment and loops, and an interpreter written in XSLT [6]. XIM programs can thus be packaged with a XIM processor (as an XSLT stylesheet) and sent to the client for execution. In this paper, we formally define the language XIM, and give a brief exposition of its operational semantics implementation in XSLT. In order to demonstrate the potential usefulness of packaging code together with its execution engine, we describe a Web application that accepts user choices regarding some computation, gets the parameters for the requested operation, generates a XIM program that does the computation, and sends the program, together with its interpreter, to the client for execution.

The implications of our approach, we believe, are significant. In principle, the programmer need not be restricted to an existing programming language on the client side, and does not need to make any assumptions about the availability of certain environments on the client side, other than that of the XSLT processor, which is already supported in all mainline Web browsers. This represents one more degree of freedom for the programmer whose code will be executed on the client side - that of choosing *any* programming language, provided its processor can be packaged and sent to the client with it.

One possible application of our idea is in the specification of the semantics of Web services. A lambda expression [7], in a suitable XML based syntax, which denotes the meaning of the service, can be sent to the client, as well as a lambda reduction machine to execute it. The client then either analyses, or executes the lambda expression to see if it fits the desired requirements. All this can be done without any computational burden on the server side.

Another application of our idea can be in the area of distributed computing, where the parameters for an operation are provided to a server, which in return generates a program corresponding to the specified operation and supplied parameters, packages the program and the processor of its language together, and sends them to the client for execution.

```
<?xml version="1.0"?>
<program>
  <vars>
      <var_declare name="fact"> 1 </var_declare>
      <var_declare name="last"> 0 </var_declare>
      <var_declare name="nb"> 5 </var_declare>
  </vars>

  <main>
       <assign varn="last">
          <var_use name="nb"/>
       </assign>
       <while>
            <condition>
                 <boolop opname="gt">
                      <var_use name="last"/>
                      <num> 1</num>
                 </boolop>
            </condition>
            <statement_list>
                 <assign varn="fact">
                      <op opname="*">
                           <var_use name="fact"/>
                           <var_use name="last"/>
                      </op>
                 </assign>

                 <assign varn="last">
                      <op opname="-">
                           <var_use name="last"/>
                           <num> 1</num>
                      </op>
                 </assign>
            </statement_list>
       </while>
       <end/> <!-- program termination -->
  </main>
</program>
```

Fig. 1. Sample XIM program for computing 5!

2 The Minimal Imperative Language XIM

We can regard a XIM program as an abstract syntax tree representation of a program written in a high-level concrete imperative language with variables of type *float*, expressions involving arithmetic operators, the usual boolean expressions, the assignment statement, one conditional construct (if-then-else) and one iteration construct (while). The formal syntax of XIM is specified using an XML Schema, given in appendix A.

XIM constructs are quite straightforward, and we just give some explanation on element attributes. Every statement element in XIM has a `@seq` attribute, which acts like the symbolic address of the element. The `@next` attribute gives the address of the next

```
var fact ← 1
var last ← 0
var nb ← 5
begin
    last ← nb
    while (last>1) do
        fact ← fact*last
        last ← last-1
    end while
end
```

Fig. 2. Pseudo-code of the XIM program for computing 5!

instruction to be executed. In <while> and <if> constructs, attributes `@true_next` (`@false_next`) determine control flow when the condition is true (false). The values of these attributes are automatically generated, using XSLT, before execution starts.

Figure 1 depicts a XIM program which computes 5!. The program demonstrates the use of *assignment* and *while* constructs in the language. The same program in pseudo-code is given in Figure 2.

3 Operational Semantics of XIM in XSLT

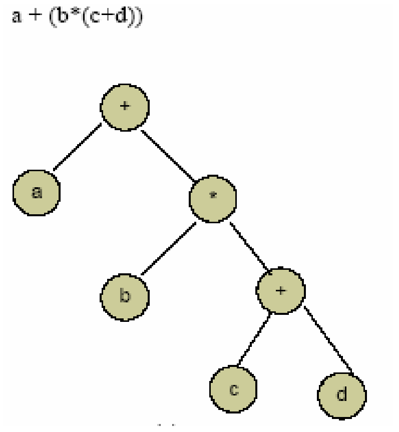
3.1 What is operational semantics?

Operational semantics specifies the meaning of a program in a source language by giving rules for how the state of an abstract, or real, well-understood machine changes for each construct of the source language. The states of the abstract machine can be represented as $< code, memory >$ pairs (called *configurations*), where $code$ represents the remaining computation, $memory$ is the current contents of the memory of the abstract machine. A transition function specifies the transition from one configuration of the machine to the next. The transition function defines the operational semantics of the language and can also act as an interpreter for it.

In the case of using XSLT for specifying the operational semantics of programming languages, a configuration consists of the memory, the program and the current value of the "program counter," all inside an XML document. The transition function is specified as XSLT templates.

3.2 Implementing Operational Semantics: The XIM Interpreter in XSLT

An XSLT stylesheet is used to implement the operational semantics of XIM. Before execution starts, the stylesheet transforms the XIM program to make it ready for interpretation. First the "program counter" is introduced as a variable in the program, and symbolic addresses are inserted into instructions as `@seq` attributes. Then the values in the `@next`, `@true_next` and `@false_next` attributes of the instructions are determined and inserted into the code. These attributes specify the address of the next instruction to execute, as in attribute grammars used in syntax directed compilation [8]. After the initialization stage, other templates are applied to the transformed XIM program to execute it. The application of templates is an iterative process, which stops when the next instruction is the `<end>` element.



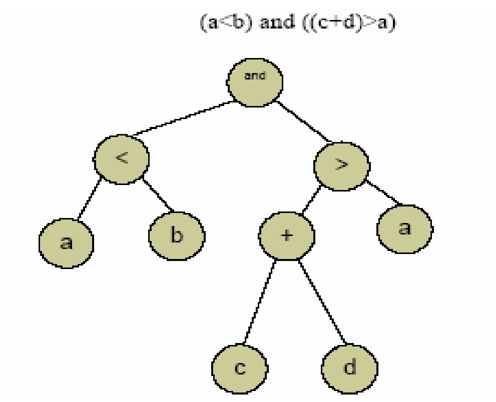
```
<op opname="+">
    <var_use name="a"/>
    <op opname="*">
        <var_use name="b"/>
        <op opname="+">
            <var_use name="c"/>
            <var_use name="d"/>
        </op>
    </op>
</op>
```

Fig. 3. An arithmetic expression and its corresponding XIM code

The top level template (Appendix B.1) of the stylesheet matches the root and performs the application of templates for initialization and interpretation. Appendix B.2 depicts the templates for the insertion of the “program counter” as a XIM variable named `PC` and symbolic addresses as `@seq` attributes. The `Sequencer` template (Appendix B.3) achieves the insertion of symbolic addresses through the use of the `<number>` element of XSLT. Values in the `@next`, `@true_next` and `@false_next` attributes are then inserted by templates that perform a case by case analysis of a given node to determine its type, inspect its position relative to other instructions, and set the control flow information as jump addresses accordingly.

The template for the implementation of the iterative steps is `Interpret` (Appendix B.4) and the template for determining the type of current instruction and calling the corresponding template for its execution is `Execute` (Appendix B.5) .

The `Assignment` template (Appendix B.6) receives an assignment statement as an XML sub-tree in the `$c` parameter. The template also has access to the XML document as a whole via the `$prg` parameter. The name of the program variable to whom a new value is to be assigned is copied into the XSLT variable `$varname`. The whole



```
<boolop opname="and">
    <boolop opname="lt">
        <var_use name="a"/>
        <var_use name="b"/>
    </boolop>
    <boolop opname="gt">
         <op opname="+">
             <var_use name="c"/>
             <var_use name="d"/>
         </op>
        <var_use name="a"/>
    </boolop>
</boolop>
```

Fig. 4. A boolean expression and its corresponding XIM code

<memory> element of the XML document is recreated to reflect the updated value of the variable, as well as the updated value of the "program counter" PC variable.

The template Construct (Appendix B.7) handles the execution of <while> and <if> statements. The instruction which is an <if> or <while> construct comes into parameter $c and the code of the program as a whole comes in parameter $prg. The truth value of the condition is determined and assigned to the XSLT variable $condition. Then the <memory> element, with all its children except the PC, is copied. The value of the PC variable is updated depending on the truth value of $condition. When it is "true," the value of the @true_next attribute of the <condition> child of the context node is assigned to the PC variable. Otherwise the value of @false_next attribute is assigned to it.

The Evaluate template (Appendix B.8), takes a parameter $n holding an XML sub-tree representing an expression. Figures 3 and 4 depict examples of arithmetic and boolean expressions respectively. This template also has access to the whole XIM program in the parameter $p. It first checks whether the incoming parameter is a numerical literal (<num>), a variable (<var use>) or a complex expression (<op> or <boolop>). If the incoming expression in parameter $n is a <num> it returns the content of the <num> element. If it is a <var_use> element (an r-value), the value of the <var declare> element referenced by <var use> is returned. If the incoming expression is an <op> or <boolop>, the type of operation required is determined from the @opname attribute. The possible values in the @opname attribute for the <op> element are *, +, -, /, intdiv, and mod, while the options for <boolop> are or, and, not, lt, gt, eq, ne, ge, le which have their conventional meanings.

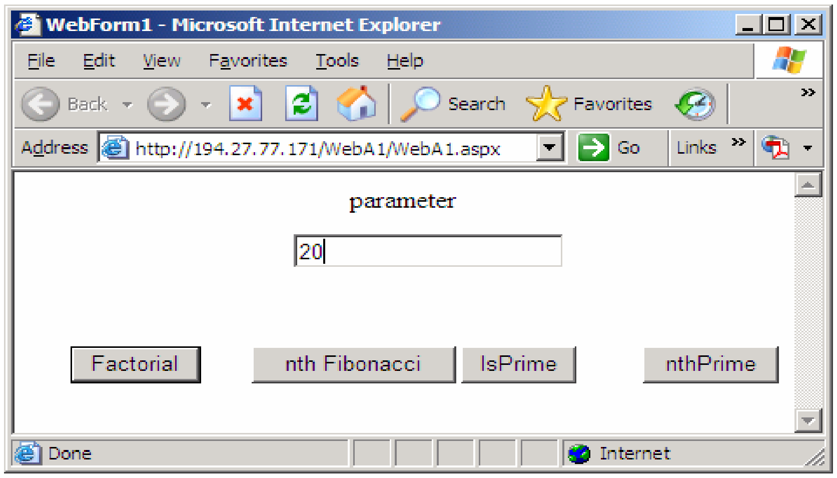

WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://194.27.77.171/WebA1/WebA1.aspx Go Links
parameter
20
Factorial
nth Fibonacci
IsPrime
nthPrime
Done
Internet

Fig. 5. The graphical user interface of the web application

Boolean expressions are evaluated fully in the same manner as arithmetic expressions (Appendix B.9).

1. XmlDocumentObject ← XML document containing the XIM code of the requested computation
2. Get the user input parameter and insert it into XmlDocumentObject
3. Insert the the processing instruction element to declare the interpreter stylesheet
4. Save the modified document in a file
5. Redirect the client to the saved XML file

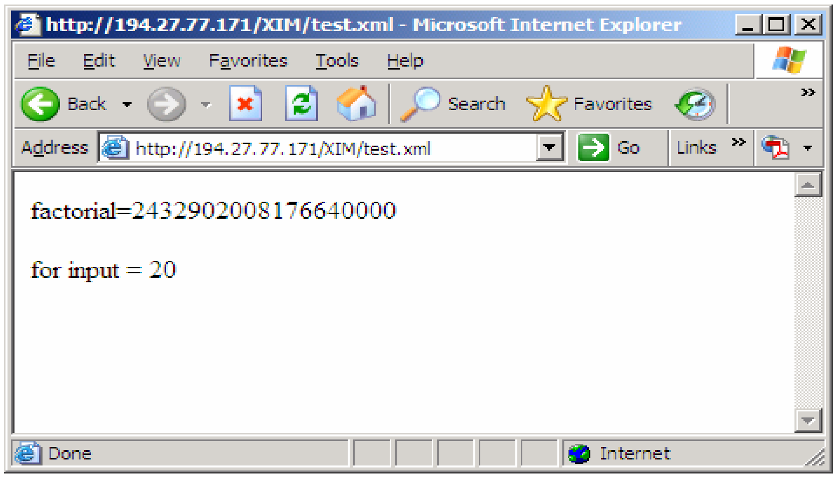
Fig. 6. High level algorithm of the Web application program

4 Sending Code for Execution on the Client: Sample Web Application

In our sample application, the user decides on a function to invoke (one of "Factorial," "Fibonacci," "IsPrime" and "nthPrime"), provides the parameter, and clicks on the button corresponding to the desired operation (Figure 5). The basic structure of the code that processes the client input is depicted in Figure 6. First, the stored "pre-packaged" XIM code for the requested computation (whose only missing part is the input parameter value) is loaded into an `XmlDocument` object and the parameter supplied by the user is inserted into the code at the correct spot. Then the processing instruction indicating the name of the stylesheet (i.e. the XIM interpreter) to be applied to the document is

```
private void FactBtn_Click(object sender, System.EventArgs e)
 {
  XmlDocument xmlDoc=new XmlDocument();
  xmlDoc.Load("c://inetpub/wwwroot/xim/fact5.xml");

  XmlElement var_n=xmlDoc.CreateElement("var_declare","");

  var_n.InnerText=TextBox1.Text;
  var_n.SetAttribute("name","nb"); // Adding the name attribute to the variable element

  //Appending the client input as a variable to the XIM code for factorial computation
  xmlDoc.DocumentElement.FirstChild.AppendChild(var_n);

  XmlProcessingInstruction newPI;//Processing instruction to declare stylesheet name
  String PItext = "type='text/xsl' href='interp.xsl'";

  //Assigning the name of the interpreter stylesheet to the processing instruction
  newPI = xmlDoc.CreateProcessingInstruction("xml-stylesheet", PItext);

  // Add the processing instruction node to the document
  xmlDoc.InsertBefore(newPI, xmlDoc.DocumentElement);

  xmlDoc.Save("c://Inetpub/wwwroot/XIM/test.xml");// Save the modifications to the XIM code

  //Redirectng the client response the just created XIM file
  Response.Redirect("http://194.27.78.64/XIM/test.xml");
 }
```

Fig. 7. Fragment of the aspx.cs code for factorial computation


http://194.27.77.171/XIM/test.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://194.27.77.171/XIM/test.xml Go Links
Done Internet

```
factorial=2432902008176640000

for input = 20
```

Fig. 8. The result of the computation of 20! executed and displayed on client's browser

inserted into the code. The resulting code in the object is stored in an XML document, and the client browser is redirected to this XML document.

As a demonstration of what happens when the client supplies a parameter and makes a choice, we give in Figure 7 the code that processes a "factorial" request. The result of the interpretation on the client machine of the factorial function, applied to number "20," is shown in Figure 8.

5 Related Work

XMILE [9] is an XML-based imperative language. Its goal is to keep mobile information devices synchronized by providing updates without stopping execution. Programs written in XMILE are transferred in source form and then interpreted on the remote host which has the interpreter implemented in Java. In XMILE's case, the interpreter is already existent/installed in the clients. In our case, the interpreter, implemented in XSLT, is bundled together with the program. [10] describes how to build interpreters for XML based scripting languages. [11] introduces ReX which is a simple reactive programming language based on XML. Superx++ [12] is an XML based object-oriented language whose runtime environment is developed using C++.

6 Conclusion

We defined an XML based imperative language "XIM," using XML Schema and implemented its operational semantics in XSLT version 1.0. This permits us to write programs in XIM, send them over the Internet to the client, and have it executed on the client through the application of the XSLT stylesheet. As a demonstration of the practicality of the approach, we developed a Web application where a user selects the desired function and provides the parameter(s) for it. The Web application sends a XIM program to the client browser, which is customized to do the computation with the provided parameter, as well as the XSLT interpreter stylesheet. The XIM program then runs on the client, doing the desired computation.

Sending code over the Internet to be executed on client Web browsers, represents one more degree of freedom than what is currently available, which is that the language available in the browsers is fixed, and only programs written in that language, and data to be used by that program, are sent to the browser. The possibility of bundling together the language processor with the program and data opens up tremendous opportunities, allowing special purpose languages to be developed and used without requiring the end users to install additional components to their Web browser. This also largely eliminates compatibility problems stemming from the unstandardized nature of scripting (and other) languages built into Web browsers currently in use today.

References

1. Java 2 Developer Team. Java 2 SDK, standard edition documentation version 1.4.2. Available at http://java.sun.com/j2se/1.4.2, 2005.

2. Microsoft .NET Developer Team. Microsoft .NET development kit. Available at http://www.microsoft.com/net/default.mspx, 2005.
3. JavaScript Developer Team. JavaScript. Available at http://www.javascript.com, 2005.
4. VBScript Developer Team. VBScript. Available at http://www.microsoft.com, 2005.
5. James Clark (Editor). XSL transformations (XSLT) version 1.0 of W3C working draft. http://www.w3.org/TR/xslt, 1999.
6. Ruhsan Onder and Zeki Bayram. Interpreting imperative programming languages in XSLT. In *Proceedings of the Ninth IASTED International conference on Internet and Multimedia Systems and Applications (EuroIMSA2005)*, pages 131–136. IASTED, 2005.
7. H. Barendregt. *The Lambda Calculus*. North-Holland, 1984.
8. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1983.
9. Cecilia Mascolo, Luca Zanolin, and Wolfgang Emmerich. XMILE: An XML based approach for incremental code mobility and update. *Automated Software Eng.*, 9(2):151–165, 2002.
10. Fabio Arjona Arciniegas. *C++/XML*, chapter 13 (Creating XML-based Extension Languages for C++ Programs). New Riders Publishing, 2001.
11. Jennifer L. Beckham, Giuseppe Di Fabbrizio, and Nils Klarlund. Towards SMIL as a foundation for multimodal, multimedia applications. In *Proceedings of EUROSPEECH-2001*, pages 1363 – 1366, 2001.
12. Kimanzi Mati. Superx++. Available at http://xplusplus.sourceforge.net/indexPage.htm, 2006.

APPENDIX

A XML Schema Describing XIM Syntax

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:attribute name="name" type="xsd:string"/>

    <xsd:attribute name="opname" type="opnameTyp"/>

    <xsd:simpleType name="opnameTyp">
       <xsd:restriction base="xsd:string">
           <xsd:enumeration value="+"/>
           <xsd:enumeration value="-"/>
           <xsd:enumeration value="*"/>
           <xsd:enumeration value="/"/>
           <xsd:enumeration value="intdiv"/>
           <xsd:enumeration value="mod"/>
           <xsd:enumeration value="and"/>
           <xsd:enumeration value="or"/>
           <xsd:enumeration value="not"/>
           <xsd:enumeration value="lt"/>
           <xsd:enumeration value="gt"/>
           <xsd:enumeration value="eq"/>
           <xsd:enumeration value="ne"/>
           <xsd:enumeration value="le"/>
           <xsd:enumeration value="ge"/>
       </xsd:restriction>
    </xsd:simpleType>


    <xsd:element name="program">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="vars" type="varsTyp" minOccurs="1" maxOccurs="1"/>
               <xsd:element name="main" type="instTyp" minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
         </xsd:complexType>
    </xsd:element>


   <xsd:complexType name="varsTyp">
       <xsd:sequence>
           <xsd:element name="var_declare" type="var_declTyp" minOccurs="0"
                                                              maxOccurs="unbounded"/>
       </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="var_declTyp">
       <xsd:simpleContent>
            <xsd:extension base="xsd:float">
                 <xsd:attribute ref="name"  use="required"/>
            </xsd:extension>
       </xsd:simpleContent>
   </xsd:complexType>


   <xsd:group name="instructions"><!--minOccurs="0" maxOccurs="unbounded"-->
       <xsd:choice>
           <xsd:element ref="assign" minOccurs="0" maxOccurs="unbounded"/>
           <xsd:element ref="if" minOccurs="0" maxOccurs="unbounded"/>
           <xsd:element ref="while" minOccurs="0" maxOccurs="unbounded"/>
           <xsd:element name="statement_list" type="instTyp" minOccurs="0"
                                                               maxOccurs="unbounded"/>
           <xsd:element name="end" minOccurs="0" maxOccurs="1">
                <xsd:complexType>
```

```
             </xsd:complexType>
         </xsd:element>
     </xsd:choice>
 </xsd:group>


 <xsd:element name="assign">
     <xsd:complexType>
         <xsd:sequence>
             <xsd:group ref="expressionKinds" minOccurs="1" maxOccurs="1"/>
         </xsd:sequence>
         <xsd:attribute name="varn" type="xsd:string" use="required"/>
     </xsd:complexType>
 </xsd:element>


 <xsd:element name="while" type="constructTyp"/>


 <xsd:element name="if">
     <xsd:complexType>
          <xsd:complexContent>
               <xsd:extension base="constructTyp">
                    <xsd:all>
                         <xsd:element name="statement_list" type="instTyp" minOccurs="0"
                                                                            maxOccurs="1"/>
                    </xsd:all>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
</xsd:element>


<xsd:complexType name="instTyp">
     <xsd:group ref="instructions" minOccurs="1" maxOccurs="unbounded"/> <!--109-->
</xsd:complexType>

 <xsd:group name="expressionKinds"> <!-- minOccurs="0" maxOccurs="2"-->
      <xsd:choice>
          <xsd:element ref="op" />
          <xsd:element name="num" type="xsd:float"/>
          <xsd:element ref="var_use"/>
      </xsd:choice>
 </xsd:group>

 <xsd:element name="var_use">
      <xsd:complexType >
           <xsd:attribute ref="name"  use="required"/>
      </xsd:complexType>
 </xsd:element>

 <xsd:element name="op" type="opTyp"/>

 <xsd:complexType name="opTyp">
          <xsd:sequence>
               <xsd:group ref="expressionKinds" minOccurs="0" maxOccurs="2"/>
          </xsd:sequence>
          <xsd:attribute ref="opname"  use="required"/>
 </xsd:complexType>


 <xsd:complexType name="constructTyp">
      <xsd:sequence>
          <xsd:element name="condition" type="condTyp" minOccurs="1" maxOccurs="1"/>
          <xsd:element name="statement_list" type="instTyp" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
 </xsd:complexType>
```

```
   <xsd:complexType name="condTyp">
          <xsd:sequence>
            <xsd:element ref="boolop" minOccurs="1" maxOccurs="1"/>
        </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="boolop">
        <xsd:complexType>
             <xsd:complexContent>
                  <xsd:extension base="opTyp">
                       <xsd:sequence>
                            <xsd:element ref="boolop" minOccurs="0" maxOccurs="2"/>
                       </xsd:sequence>
                  </xsd:extension>
             </xsd:complexContent>
        </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

B Code Fragments Of The Interpreter Stylesheet

B.1 Top-level template that matches the root, applies all other templates for initializing and interpretation, and finally displaying the result

```
<xsl:template match="/">
     <xsl:variable name="prgvar">
          <prog>
              <xsl:apply-templates/> <!--apply to all children of root -->
          </prog>
     </xsl:variable> <!-- PC and @seq inserted -->


     <xsl:variable name="prgnext">
        <prog>
           <xsl:copy-of select="msxsl:node-set($prgvar)/prog/memory"/>
           <main>
              <xsl:for-each select="msxsl:node-set($prgvar)/prog/main/child::node()">
                    <xsl:call-template name="Seq2">
                         <xsl:with-param name="node" select="."/>
                    </xsl:call-template>
              </xsl:for-each> <!-- @next inserted -->
           </main>
        </prog>
     </xsl:variable>


     <xsl:variable name="RM">
         <xsl:call-template name="Interpret">
             <xsl:with-param name="prg">
                  <xsl:copy-of select="msxsl:node-set($prgnext)/prog/memory"/>
                  <xsl:copy-of select="msxsl:node-set($prgnext)/prog/main"/>
             </xsl:with-param>
         </xsl:call-template> <!-- memory is sent from interpret-->
     </xsl:variable>

     <html>
        <head> Result of execution </head>
        <body>
           <xsl:for-each select="msxsl:node-set($RM)/child::node()[1]/child::node()">
                <xsl:if test="count(./preceding-sibling::*)=0">
                    <xsl:value-of select="@name"/>
                    <xsl:value-of select="’=’"/>
                    <xsl:value-of select="."/>
                    <p/>
                </xsl:if>

                <xsl:if test="count(./following-sibling::*)=1">
                    <xsl:value-of select="’  for input = ’"/>
                    <xsl:value-of select="."/>
                </xsl:if>
           </xsl:for-each>
        </body>
     </html>
</xsl:template>
```

B.2 The templates for inserting the program counter and calling the `Sequencer` template

```
<xsl:template match="program/vars">
    <memory>
        <xsl:for-each select="./child::node()">
            <xsl:copy-of select="."/>
        </xsl:for-each>
        <xsl:element name="var_declare">
```

```
            <xsl:attribute name="name">
                <xsl:value-of select="’PC’"/>
            </xsl:attribute>

            <xsl:value-of select="’1’"/>
        </xsl:element>
    </memory>
</xsl:template>

<xsl:template match="program/main">
    <main>
       <xsl:for-each select="./child::node()">
           <xsl:call-template name="Sequencer">
               <xsl:with-param name="node" select="."/>
           </xsl:call-template>
       </xsl:for-each>
    </main>
</xsl:template>
```

B.3 Code For Sequencer Template

```
<xsl:template name="Sequencer">
    <xsl:param name="node"/>
    <xsl:variable name="nm">
         <xsl:number level="any"  format="1" count="program/main//child::*"/>
    </xsl:variable>

 <xsl:choose>
        <xsl:when test="name($node)=’assign’">
            <xsl:copy>
                <xsl:attribute name="varn">
                    <xsl:value-of select="$node/@varn"/>
                </xsl:attribute>
                <xsl:attribute name="seq">
                    <xsl:value-of select="$nm"/>
                </xsl:attribute>

                <xsl:for-each select="$node/child::node()">
                    <xsl:call-template name="Sequencer">
                        <xsl:with-param name="node" select="."/>
                    </xsl:call-template>
                </xsl:for-each>
             </xsl:copy>
          </xsl:when>
          <xsl:when test="name($node)=’while’ or name($node)=’if’ or
                                                           name($node)=’statement_list’">
              <xsl:copy>
                     <xsl:attribute name="seq">
                         <xsl:value-of select="$nm"/>
                     </xsl:attribute>

                     <xsl:for-each select="$node/child::*">
                         <xsl:call-template name="Sequencer">
                             <xsl:with-param name="node" select="."/>
                         </xsl:call-template>
                     </xsl:for-each>
               </xsl:copy>
           </xsl:when>
           <xsl:when test="name($node)=’end’">
              <xsl:copy>
                     <xsl:attribute name="seq">
                         <xsl:value-of select="$nm"/>
                     </xsl:attribute>
               </xsl:copy>
           </xsl:when>
           <xsl:otherwise>
               <xsl:copy-of select="$node"/>
```

```
       </xsl:otherwise>
     </xsl:choose>
</xsl:template>
```

B.4 The `Interpret` template for providing the iterative application of templates

```
<xsl:template name="Interpret">
    <xsl:param name="prg"/>

    <xsl:variable name="SeqOfCInst"
                  select="msxsl:node-set($prg)/memory/var_declare[@name='PC']"/>
    <xsl:variable name="CurrentInst"
                  select="msxsl:node-set($prg)/main//child::*[@seq=$SeqOfCInst]"/>

    <xsl:choose>
        <xsl:when test="name($CurrentInst)='end'">
            <xsl:copy-of select="msxsl:node-set($prg)/memory"/>
        </xsl:when>

        <xsl:otherwise>
             <xsl:variable name="TempMRes">
                  <xsl:call-template name="Execute">
                      <xsl:with-param name="currentel" select="$CurrentInst"/>
                      <xsl:with-param name="prgnd" select="msxsl:node-set($prg)"/>
                  </xsl:call-template>
             </xsl:variable>


             <xsl:call-template name="Interpret">
                  <xsl:with-param name="prg">
                      <xsl:copy-of select="msxsl:node-set($TempMRes)/memory"/>
                      <xsl:copy-of select="msxsl:node-set($prg)/main"/>
                  </xsl:with-param>
             </xsl:call-template>
        </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

B.5 Code of template `Execute`

```
<xsl:template name="Execute">
    <xsl:param name="currentel"/>
    <xsl:param name="prgnd"/>

    <xsl:choose>
        <xsl:when test="name($currentel)='assign'">
            <xsl:call-template name="assignment">
                <xsl:with-param name="prg" select="$prgnd"/>
                <xsl:with-param name="c" select="$currentel"/>
               <!--parameter holding the assignment node with all its children-->
            </xsl:call-template>
        </xsl:when>

        <xsl:when test="name($currentel)='if' or name($currentel)='while'">
            <xsl:call-template name="Construct">
                <xsl:with-param name="c" select="$currentel"/>
                <xsl:with-param name="prg" select="$prgnd"/>
            </xsl:call-template>
       </xsl:when>

        <xsl:when test="name($currentel)='statement_list'">
            <xsl:call-template name="Execute">
                <xsl:with-param name="currentel" select="$currentel/child::*[1]"/>
                <!--the first child of statement list-->
                <xsl:with-param name="prgnd" select="$prgnd"/>
            </xsl:call-template>
        </xsl:when>

        <xsl:otherwise> <!-- end -->
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

B.6 Template `Assignment` to handle assignment operation

```
<xsl:template name="Assignment">
    <xsl:param name="c"/>
    <xsl:param name="prg"/>

    <xsl:variable name="varname">
        <xsl:value-of select="$c/@varn"/>
    </xsl:variable> <!--the name of the variable at the left hand side of the assignment-->

    <xsl:element name="memory">
        <xsl:for-each select="msxsl:node-set($prg)/memory/child::*[@name!='PC']">
            <xsl:choose>
                <xsl:when test="@name!=$varname">
                    <xsl:copy-of select="."/>
                </xsl:when>

                <xsl:otherwise><!-- if matches-->
                    <xsl:variable name="expr">
                        <xsl:call-template name="Evaluate">
                            <xsl:with-param name="n" select="$c/child::*[1]"/>
                            <xsl:with-param name="p" select="msxsl:node-set($prg)"/>
                        </xsl:call-template>
                       <!-- evaluate the op to get the value of assigned expression-->
                    </xsl:variable>

                    <xsl:element name="var_declare">
                        <xsl:attribute name="name">
                            <xsl:value-of select="$varname"/>
                        </xsl:attribute>
                        <xsl:value-of select="number($expr)"/> <!--new(assigned) value -->
                    </xsl:element>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>

        <xsl:element name="var_declare">
            <xsl:attribute name="name">
                <xsl:value-of select="'PC'"/>
            </xsl:attribute>
            <xsl:value-of select="$c/@next"/>
       </xsl:element>
    </xsl:element>   <!--memory-->
</xsl:template>
```

B.7 Template `Construct` to handle “While” and “If-then-else” constructs

```
<xsl:template name="Construct">
    <xsl:param name="c"/> <!-- current node/instruction -->
    <xsl:param name="prg"/>

    <xsl:variable name="condition">
        <xsl:call-template name="Evaluate">
            <xsl:with-param name="n" select="$c/condition/child::*[1]"/>
            <xsl:with-param name="p" select="msxsl:node-set($prg)"/>
       </xsl:call-template>
    </xsl:variable>
    <xsl:element name="memory">
        <xsl:for-each select="msxsl:node-set($prg)/memory/child::*[@name!='PC']">
            <xsl:copy-of select="."/><!--copy all vars -->
        </xsl:for-each>
        <xsl:element name="var_declare">
            <xsl:attribute name="name">
                <xsl:value-of select="'PC'"/>
            </xsl:attribute>
            <xsl:choose>
                <xsl:when test="$condition='true'">
                    <xsl:value-of select="$c/condition/@true_next"/>
                </xsl:when>
                <xsl:otherwise> <!-- condition=0 (false) -->
                    <xsl:value-of select="$c/condition/@false_next"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>
    </xsl:element> <!--memory-->
</xsl:template>
```

B.8 Code fragment showing a part of template `Evaluate` for evaluating arithmetic expressions

```
<xsl:template name="Evaluate">
    <xsl:param name="n"/>
    <xsl:param name="p"/>

    <xsl:if test="name($n)='num'">
        <xsl:value-of select="number($n)"/>
    </xsl:if>

    <xsl:if test="name($n)='var_use'">
        <xsl:variable name="varname">
            <xsl:value-of select="$n/@name"/>
        </xsl:variable> <!-- get the name of the variable to be used-->

        <xsl:for-each select="$p/memory/child::*">
            <xsl:if test="@name=$varname">
                <xsl:value-of select="number(.)"/>
            </xsl:if>
        </xsl:for-each>
    </xsl:if>

    <xsl:if test="name($n)='op'">
        <xsl:choose>
            <xsl:when test="$n/@opname='*'" >
                <xsl:variable name="c1">
                    <xsl:call-template name="Evaluate">
                        <xsl:with-param name="n" select="$n/child::*[1]"/>
                        <xsl:with-param name="p" select="$p"/>
                    </xsl:call-template>
                </xsl:variable>

                <xsl:variable name="c2">
                    <xsl:call-template name="Evaluate">
                        <xsl:with-param name="n" select="$n/child::*[2]"/>
                        <xsl:with-param name="p" select="$p"/>
                    </xsl:call-template>
                </xsl:variable>
                <xsl:value-of select="number($c1)*number($c2)"/>
            </xsl:when>
                          ................
```

B.9 Code fragment from template `Evaluate` that handles boolean expressions

```
<xsl:if test="name($n)='boolop'">
       <xsl:choose>
           <xsl:when test="$n/@opname='or'">
               <xsl:variable name="c1">
                   <xsl:call-template name="Evaluate">
                       <xsl:with-param name="n" select="$n/child::*[1]"/>
                       <xsl:with-param name="p" select="$p"/>
                   </xsl:call-template>
               </xsl:variable>

               <xsl:variable name="c2">
                   <xsl:call-template name="Evaluate">
                       <xsl:with-param name="n" select="$n/child::*[2]"/>
                       <xsl:with-param name="p" select="$p"/>
                   </xsl:call-template>
               </xsl:variable>

               <xsl:value-of select="($c1='true') or ($c2='true')"/>
           </xsl:when>

           <xsl:when test="$n/@opname='not'">
               <xsl:variable name="c1">
                   <xsl:call-template name="Evaluate">
                       <xsl:with-param name="n" select="$n/child::*[1]"/>
                       <xsl:with-param name="p" select="$p"/>
                   </xsl:call-template>
               </xsl:variable>

               <xsl:value-of select="$c1='false'"/>
           </xsl:when>

           <xsl:when test="$n/@opname='lt'">
               <xsl:variable name="c1">
                   <xsl:call-template name="Evaluate">
                       <xsl:with-param name="n" select="$n/child::*[1]"/>
                       <xsl:with-param name="p" select="$p"/>
                   </xsl:call-template>
               </xsl:variable>

               <xsl:variable name="c2">
                   <xsl:call-template name="Evaluate">
                       <xsl:with-param name="n" select="$n/child::*[2]"/>
                       <xsl:with-param name="p" select="$p"/>
                   </xsl:call-template>
               </xsl:variable>

               <xsl:value-of select="number($c1) &lt; number($c2)"/>
           </xsl:when>
         ......................
      </xsl:choose>
  </xsl:if>
```