Extensible Stylesheet Language Transformations (XSLT) as a Delivery Medium for Executable Content Over Internet

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

Department of Computer Engineering/Internet Technologies Research Center
Eastern Mediterranean University
Famagusta, Cyprus
{ruhsan.onder, zeki.bayram}@emu.edu.tr
http://www.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, which is ideal for manipulating XML documents. 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

Sending code to be executed in a remote computer is a routine affair. However, the code author needs to make certain 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 that 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.

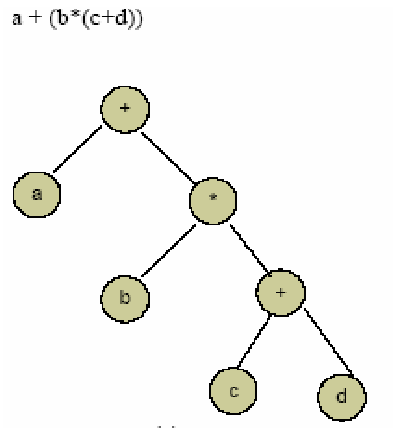
Browsers have another kind of language processor installed on them - that for Extensible Stylesheet Language Transformations XSLT [5]. XSLT stylesheets are applied to XML documents, mainly for the purpose of generating XHTML pages for presentation on browsers. Thus, XSLT helps to separate content (inside 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. In [6] we described an XML based programming language, called XIM, with imperative control features such as assignment and loops, and an interpreter for it in XSLT.

In this paper, we formally define the language XIM and give a detailed exposition of its operational semantics implementation in XSLT. In order to demonstrate the potential usefulness of this approach, 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 availabilty 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 langauge, 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 a Web service. 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. In fact, this would correspond to a new paradigm of computing, which could be called “local web services.”



```
<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>
```

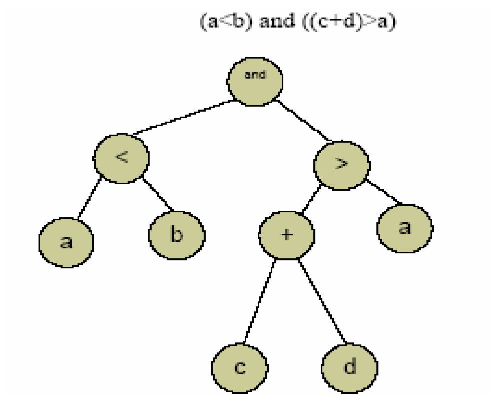
Fig. 1. An arithmetic expression and its corresponding XIM code

The remainder of this paper is organized as follows. In section 2 we informally describe the syntax of XIM and give an example XIM program. The implementation of the operational semantics of XIM in XSLT is described in section 3. This is followed in section 4 by the explanation of the web application that generates XIM code for a requested operation, packages it with the XIM interpreter, and sends them both to the client for execution. In section 5 is a discussion of advantages and disadvantages of the proposed approach, as well as the major issues involved in its implementation. Section 6 gives an overview of related research and in section 7 we have the conclusion and future research directions. Finally, appendix A contains the complete, formal syntax specification of XIM as a W3C Schema, and appendix B contains the code for the XSLT templates performing initializations.

2 The Minimal Imperative Language XIM

2.1 Overall Program Structure

XIM is a simple 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). We can regard a XIM program as an abstract syntax tree representation of a program written in a high-level concrete imperative language with similar features. A program in XIM is structured under a <`program`> root element, which in turn consists of the elements <`vars`> and



```
<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. 2. A boolean expression and its corresponding XIM code

```
<?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. 3. Sample XIM program for computing 5!

```
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. 4. Pseudo-code of the XIM program for computing 5!

<main>. The <vars> element encapsulates the variable declarations and the <main> element encapsulates the statements of the program.

2.2 Expressions

The declarations of the variables to be used in the program are represented by <var_declare> elements having a @name attribute for the name of the variable. The instructions that can be inside the <main> element are <assign> , <if> , <while> and <end> elements. The <end> element marks the termination of the program. Numeric literals are represented with the <num> element. The <var_use> element which represents the "rvalue" of a variable has attribute @name for the name of the variable whose value is to be returned in an expression. An <op> element has attribute @opname to specify the arithmetic operation it represents. Boolean expressions are represented with the <boolop> element and are similar in concept to <op> elements. Figure 1 depicts an arithmetic expression, its syntax tree and corresponding XIM code. In Figure 2 we have a boolean expression, its syntax tree, and corresponding XIM code.

2.3 Statements

In an <assign> element, the name of the variable to which a new value is to be assigned is given in the attribute @varn. The <while> instruction consists of two sub-elements; the <condition> element representing a boolean expression and a <statement_list> element enclosing the instructions to be executed as long as the condition is satisfied. The <if> instruction consists of one <condition> element and one or two <statement_list> elements , the second, if present, representing the "else" case. Every statement element in XIM has a @seq attribute, which acts like the symbolic address of the element. There is also a @next attribute which gives the address of the next instruction to be executed. In <while> and <if> constructs, in addition to the @next attribute, attributes @true_next and @false_next are used in their <condition> child, which shows the sequence number of the instruction to be executed next when the condition is true, and when the condition is false respectively. These attributes are necessary for the execution of the XIM program but the programmer does not have to deal with these "implementation details"- they are automatically generated, using XSLT, by the XSLT stylesheet before execution starts. For declaring the end of a program, the <`end`> element is used as a termination marker.

We give a sample XIM program in Figure 3 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 4.

2.4 Formal Specification of XIM syntax using XML Schema

In appendix A we give the formal syntax specification of XIM as an XML Schema. We note that the rich notation and facilities available in XML Schema more than suffice to describe the syntax of XIM and similar languages.

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. A syntax-directed translation scheme [8] is one way for specifying the operational semantics of a language, where the production rules of an attribute grammar and the actions for each production rule specify the translation of each construct in the source language.

It is also possible to more formally specify the operational semantics of a programming language by representing the states of an abstract machine as $< code, memory >$ pairs (which we can call *configurations*), where $code$ represents the remaining computation, $memory$ is the current contents of the memory of the abstract machine, and we define a function which specifies the transitions from one configuration 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. Each application of a template corresponds to a move from one configuration of the *abstract machine* to the next. The templates are applied repetitively to each intermediate state until a configuration representing a final state is reached (or forever if the program loops).

3.2 Implementing Operational Semantics: The XIM Interpeter in XSLT

Overview of the XIM Interpreter An XSLT stylesheet is used to implement the operational semantics of XIM. Before the execution, 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 sequence numbers, which shall function as 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 instructions. These attributes specify the address of the next instruction

```
<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>
```

Fig. 5. Top-level template that matches the root, applies all other templates for initializing and interpretation, and finally displaying the result

```
<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>
```

Fig. 6. The templates for inserting the program counter and calling the `Sequencer` template

```
<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>
```

Fig. 7. The Interpreter template for XIM

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, and at each iteration the current XML document represents a configuration, as described above. The iteration stops when the next instruction is the `<end>` element.

```
<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>
```

Fig. 8. Code of template `Execute`

The Top-Level Template Figure 5 depicts the code of the top level template of the stylesheet which matches the root and performs the application of templates for initializing and interpretation. Initially, the root is matched and the first set of initializer templates are applied (lines 2-6), which introduce the "program counter" and insert symbolic addresses to instructions. The result of this transformation is kept in the variable `$prgvar` as a result tree fragment, and is used to apply the second set of initializer templates (lines 7-18), which insert the "goto" attributes of instructions. The program is now ready for interpretation. The result of this last transformation is kept in the result tree fragment `$prgnext` on which the templates implementing the instructions of the program are applied (lines 19-26). Finally, the result of the computation, as stored in the program variables, is displayed as a Web page (lines 27-43).

Insertion of the "program counter" and symbolic addresses Figure 6 depicts the templates for the insertion of "program counter" as a XIM variable named `PC` (lines 1-13) and symbolic addresses as `seq` attributes (lines 14-22). The <`memory`> element is generated and the variables in the original XIM code are copied into it (lines 3-5). Then, the XIM variable `PC` is generated and inserted in the <`memory`> element (lines 6-11), and is initialized to `1` , the address of the first instruction (line 10). The `Sequencer` template, whose code is given in Appendix B, is called for the insertion of sequential addresses (lines 16-20) into all the instructions under the `main` element. The `Sequencer` template achieves this through the use of the <`number`> element of XSLT, which outputs the sequence number of a node in a specified format.

Inserting control flow information into XIM code Control flow information is inserted into XIM instructions (by a named template) in the form of `@next` , `@true_next` and `@false_next` attributes and their values. These values are symbolic addresses, of the next instruction, according to the control flow implied by instructions. The code for this task is rather long, and we have omitted it from the paper. Very briefly, it performs a case by case analysis of a given node to determine its type, inspects its position relative to other instructions, and sets the jump addreses accordingly. The full code of the application, including all the XSLT templates, is available at [9].

Note that insertion of control flow information is a "compile-time" activity, and is different from "determining the next instruction to execute" at "run-time" which is explained in the next subsection.

XSLT code implementing the Iterative Interpretation Steps The template `Interpret` given in Figure 7 takes as a parameter a XIM program, which is initialized with symbolic addresses and "goto" information for each instruction (line 2). It extracts from this program the current value of the program counter (line 3), and the instruction at the address specified by the program counter (line 4). If the instruction to execute is the <`end`> statement, `Interpret` returns back to its caller (lines 6-8), i.e. the top-level template. Otherwise, the template `Execute` is called to carry out the actions required by the current instruction (lines 10-15). When this is finished, `Interpret` is called again, recursively, but with a new configuration (lines 16-21). This amounts to iteratively applying the `Execute` template until the <`end`> statement is reached (depending on the program, of course, this may never happen!).

The `Execute` template The `Execute` template in Figure 8 determines the type of its argument and calls the correct template accordingly. The `Assignment` template is called in lines 5-10. For <`if`> and <`while`> constructs it calls the `Construct` template (lines 11-16) and for <`statement_list`> elements it calls itself recursively with the first instruction (first child) in <`statement_list`> as a parameter (17-22). This works fine, since all instructions already have in them the symbolic address of the next instruction to "jump" to. In the case of "if" and "while" statements, the adress of the next instruction is known for both "true" or "false" values of the condition as well.

For determining the type of the instruction, the XSLT <`choose`> construct is used. Once the template that executes the instruction is done, control is returned back to the

```
<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>
```

Fig. 9. Template `Assignment` to handle assignment operation

`Execute` template, which in turn returns control to its caller (the `Interpret` template). The only changes that are made to the current version of the document at run-time are the contents of the `PC` variable (the “program counter”), and possibly other variables.

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

4.      <xsl:variable name="condition">
5.          <xsl:call-template name="Evaluate">
6.              <xsl:with-param name="n" select="$c/condition/child::*[1]"/>
7.              <xsl:with-param name="p" select="msxsl:node-set($prg)"/>
8.         </xsl:call-template>
9.      </xsl:variable>

10.     <xsl:element name="memory">
11.         <xsl:for-each select="msxsl:node-set($prg)/memory/child::*[@name!='PC']">
12.             <xsl:copy-of select="."/><!--copy all vars -->
13.         </xsl:for-each>
14.         <xsl:element name="var_declare">
15.             <xsl:attribute name="name">
16.                 <xsl:value-of select="'PC'"/>
17.             </xsl:attribute>
18.             <xsl:choose>
19.                 <xsl:when test="$condition='true'">
20.                     <xsl:value-of select="$c/condition/@true_next"/>
21.                 </xsl:when>
22.                 <xsl:otherwise> <!-- condition=0 (false) -->
23.                     <xsl:value-of select="$c/condition/@false_next"/>
24.                 </xsl:otherwise>
25.             </xsl:choose>
26.         </xsl:element>
27.     </xsl:element> <!--memory-->
28. </xsl:template>
```

Fig. 10. Template `Construct`to handle ”While” and ”If-then-else” costructs

The `Assignment` template The `Assignment` template is depicted in Figure 9. It receives an assignment statement (line 2) 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 (line 3).

The name of the program variable to whom a new value is to be assigned is copied into the XSLT variable `$varname` (lines 4-6). The whole <`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 (lines 7-35). Since the program variables in memory are held in <`var_declare`> elements, a new instance for each such element needs to be created with the unchanged ones having their previous values, and the updated one having its new value.

For the recreation of the <`memory`> element, the case of variables other than the “program counter” (lines 8-28) and the recreation of the “program counter” (29-34) are handled separately. All the memory elements, which are in fact variables of the XIM program, different from the one that is on the left hand side of the assignment statement are copied as-is to the new XML document (lines 10-12). For the variable whose name

```
1.  <xsl:template name="Evaluate">
2.      <xsl:param name="n"/>
3.      <xsl:param name="p"/>

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

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

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

18.     <xsl:if test="name($n)='op'">
19.         <xsl:choose>
20.             <xsl:when test="$n/@opname='*'" >
21.                 <xsl:variable name="c1">
22.                     <xsl:call-template name="Evaluate">
23.                         <xsl:with-param name="n" select="$n/child::*[1]"/>
24.                         <xsl:with-param name="p" select="$p"/>
25.                     </xsl:call-template>
26.                 </xsl:variable>

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

Fig. 11. Code fragment showing a part of template `Evaluate`for evaluating arithmetic expressions

matches the name in `$varname` , the expression (first child of the incoming assign node, found by `''$c/child::*[1]''`) on the right hand side of the assignment statement is evaluated, its result is placed in a temporary XSLT variable `$expr` (lines 14-19), and the element representing the affected variable is recreated with the new value (lines 20-25).

The value of the program counter is then modified by the recreation of a new memory element having name `PC` , whose value comes from the `@next` attribute of the current instruction (lines 29-34). This step is present in the execution of all statements, where the program counter is made ready for the next application of the operational semantics which is provided by the recursion of template `Interpret` .

Execution of the "If-then-else" and "While" constructs Figure 10 depicts the code segment for executing <`while`> and <`if`> statements. The instruction which is an <`if`> or <`while`> construct comes into parameter `$c` (line 2) and the code of the program as a whole comes in parameter `$prg` (line 3) when the template `Construct` is called. The truth value of the condition is determined then (lines 5-8) and assigned to the XSLT variable `$condition` . Then the whole <`memory`> element with all its children, except the `PC,` is copied (lines 11-13). The value of the `PC` variable is updated depending on the truth value of `$condition` (lines 14-26). When it is "true," the value of the `@true_next` attribute of the <`condition`> child of the context node is

```
 <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>
```

Fig. 12. Code fragment from template `Evaluate`that handles boolean expressions

assigned to the PC variable (lines 19-21). Otherwise the value of @false_next attribute is assigned to it (lines 22-24).

Upon the return of the Assignment or Construct to Execute , it in turn returns to its caller Interpret (of Figure 7) where the result of the call to Execute is stored in XSLT variable $TempMRes as a temporary result tree (lines 10-15). The variable $TempMRes in fact is a tree fragment holding the new memory contents after the modifications done by the instruction executed. Therefore the memory element enclosed by it is used to construct the memory element (line 18) of the new program body, together with the original <main> part, which is unchanged and has instruction elements (line 19). The new constructed program body is sent as the parameter $prg to template Interpret in its recursive call (lines 17-20).

Evaluation of Boolean and Arithmetic Expressions The Evaluate template, part of which is given in Figure 11, takes a parameter $n holding an XML sub-tree representing an expression. 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 (lines 4-6). If it is a <var_use> element, the value of the <var_declare> element referenced by <var_use> is returned (lines 7-17). 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.

For the evaluation of an <op> element, two XSLT variables are created, one for each of its children. Template Evaluate is then called recursively on these children and the return value is assigned to the newly created XSLT variables for each child. The code for the multiplication operation is depicted in the fraction of Figure 11 (lines 20-34). The return values of the recursive calls are assigned to variables $c1 and $c2 respectively. Then the XSLT operator corresponding to the one specified in the @opname attribute of <op> element (in this case, multiplication) is applied to $c1 and $c2 and the result is returned.

Boolean expressions are evaluated fully in the same manner as arithmetic expressions. Part of the code for evaluating boolean expressions is given in Figure 12.

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

4.1 Overview

The code for the Web application is developed as an ASP.NET code-behind application using Visual Studio .NET environment and C#. The graphical user interface of the application is shown in Figure 13.

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

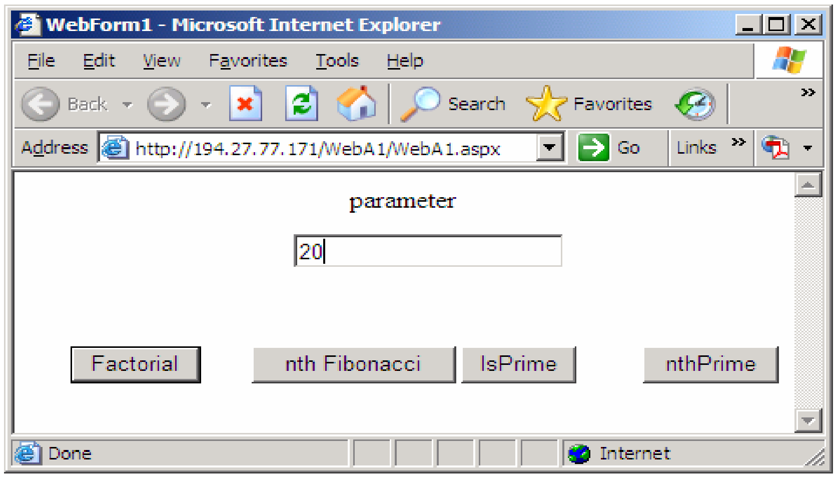

WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://194.27.77.171/WebA1/WebA1.aspx Go Links
parameter
20
Factorial
nth Fibonacci
IsPrime
nthPrime
Done
Internet

Fig. 13. The graphical user interface of the web application

the desired operation. The basic structure of the code that processes the client input is depicted in Figure 14. First, the stored "pre-packaged" XIM code for the requested computation is loaded into an XmlDocument object. The only missing part of the code is the parameter supplied by the user. This parameter is inserted into the code in this object 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 inserted to the code. The resulting code in the object is stored in an XML document, and the client browser is redirected to this XML document.

```
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. 14. High level algorithm of the Web application program

4.2 Server-side Code for Processing a "factorial" Request

As a demonstration of what happens when the client makes a choice and clicks a button, we describe the code, given in Figure 15, that processes a "factorial" request.

The code first loads the XIM code of the requested computation, which is stored at the server as an XML document, into an `XmlDocument` object (lines 3-4). The code of the factorial function is given in Figure 16. It, then creates an element to be inserted in the memory section of the code (line 5) as a XIM variable. The value of the parameter entered by the client is taken and inserted as *innerText* (line 6). In line 7, the newly created element is given an attribute called `name` and its value is set to `nb` , which is the name of the variable that will contain the parameter value. The element is inserted into memory by appending it to the first child of the document element of `XmlDocument` object (line 8). In order to indicate the name of the stylesheet which will be used for transformation, an `XmlProcessingInstruction` object is created with the name of the interpreter stylesheet (lines 9-11) and it is appended to the XIM code by inserting it before the document element of the `XmlDocument` object (line 12). The modified XIM code, is stored by saving the contents of the `XmlDocument` object in a file (line 13) and then the response to client is redirected to that file (line 14).

By the redirection of the client response, the interpreter stylesheet is applied to the modified XIM code by the XSLT processor of the client browser. The result of the interpretation on the client machine of the factorial function, applied to nuumber “20”, is shown in Figure 17.

```
1.  private void FactBtn_Click(object sender, System.EventArgs e)
2.   {
3.    XmlDocument xmlDoc=new XmlDocument();
4.    xmlDoc.Load("c://inetpub/wwwroot/xim/fact5.xml");

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

6.    var_n.InnerText=TextBox1.Text;
7.    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
8.    xmlDoc.DocumentElement.FirstChild.AppendChild(var_n);

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

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

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

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

      //Redirectng the client response the just created XIM file
14.   Response.Redirect("http://194.27.78.64/XIM/test.xml");
15.  }
```

Fig. 15. Fragment of the aspx.cs code for factorial computation

5 Discussion

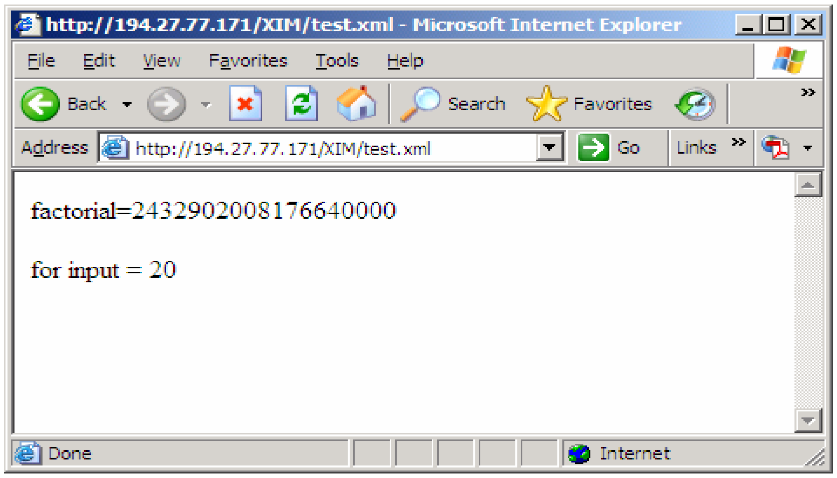
Although newer constructs of XSLT such as variables, named templates and parameters provide a lot of support for “programming” (especially through the recursive usage of named templates), it has proved challenging to implement the operational semantics of an imperative language using XSLT. Specifically, keeping state change updates through the use of XSLT variables is not possible, because variables in XSLT are *single assignment*, meaning that an XSLT variable gets its value only at creation time, and its value cannot be changed later on. Another difficulty was that contents of parameters and

```
<?xml version="1.0"?> <program>
  <vars>
      <var_declare name="fact"> 1 </var_declare>
      <var_declare name="last"> 0 </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. 16. XIM program which the web application uses for computing the factorial function (the variable "nb" is missing, and will be inserted by the application after the value is obtained from the user)


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

variables in XLST are regarded as “result tree fragments,” which are not treated as regular node sets. Therefore, templates in an XSLT stylesheet cannot be applied to XML components generated by the stylesheet itself when the content is stored in an XSLT variable (this shortcoming of XSLT version 1.0 has been remedied in XSLT version 2.0 [10] by the introduction of “temporary trees,” but we have not used XSLT version 2.0 because it is not yet supported by current browsers). We thus had to use the “node-set()” extension function of Microsoft’s MSXML [11], which converts “result tree fragments” into node sets.

6 Related Work

Research on XML technologies has resulted in a number of XML based languages for various purposes. However, our literature search leads us to believe that our approach of using XSLT for the specification of the operational semantics of an imperative programming language and delivery of executable content is quite novel.

Reported work that most closely resembles ours is XMILE [12]. XMILE is an XML-based imperative language which follows the line of XPL
[13]. It is an XML based scripting language which provides code mobility and incremental updates at a very fine-grained level for applications requiring a more flexible approach than what can be achieved with Java and mobile agents. Its goal is to keep mobile information devices synchronized by providing updates without stopping execution. Unlike Java programs that are sent in compiled form, programs written in XMILE are transferred in source form and then interpreted on the remote host *which has the interpreter* implemented in Java for the code. Incremental updates to the code are realized by the insertion or substitution of small code fragments. Their approach resembles ours in that XML-based imperative languages are used and their interpreters are provided. 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.

[14] describes how to build interpreters for XML based scripting languages. [15] introduces ReX which is a simple reactive programming language based on XML. ReX provides an expressive thread control and event system to augment SMIL [16] for providing more expressive multimodal multimedia programming. [17] describes a multi-threaded, XML-based scripting language ReaX which enhances the ideas of ReX. It provides a reactive notation that captures notions of events, event notifications, real-time scheduling, and simultaneity. Wireless Markup Language (WML) [18] is an XML-defined markup language for creating and serving up wireless-friendly information and applications for WAP (Wireless Application Protocol) enabled devices and browsers. VoiceXML [19] is a markup language designed to make Internet content and information accessible via voice and phone. X3D [20] is an XML-enabled 3D file format to enable real-time communication of 3D data across all applications and network applications. It has a rich set of features for use in engineering and scientific visualization,multimedia and entertainment. VEML [21] is a mark up language to describe web-based virtual environments through atomic simulations. VEML can be used to create and/or to extend highly dynamic 3D virtual environments independent from a support environment without being closely tight to a specific application. MathML [22] is a

low-level specification for describing mathematics as a basis for machine to machine communication. XPointers [23] is designed to address the problem of URL which only points at a single, complete document. XPointers provides addressing in a more granular level, such as linking to the third sentence of the seventeenth paragraph in a document which otherwise requires the author of the targeted document to manually insert named anchors at the targeted location. The author of the document doing the linking cannot do this unless he or she also has write access to the document being linked to. ebXML [24] aims to provide standard specifications for business processes, core data components, collaboration protocol agreements, messaging, registries and repositories to enable business over Internet. Superx++ [25] is an XML based object-oriented language whose runtime environment is developed using C++.

The works of [26] and [27] are two representative approaches on the usage of XML/XSLT for security implementations. [26] describes an XML based encryption framework which enables users of third party distribution systems to query over encrypted data. All confidentiality and authenticity information are encoded in XML and attached to the encoded data by the owner. [27] presents an XML document security model and a language DSL to implement the operational model. For executing DSL specifications they provide two approaches and compare them. The first of the approcahes is the development of XSLT extension functions in Java to enable XSLT to implement the DSL processing model via transformations. This bears similarities to our approach of using XSLT as a language processor. The second approach is the implementation of the processing model in Java and using an XML parser for Java, instead of using XSLT.

7 Conclusion and Future Research Directions

We defined an XML based imperative language, called "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 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 also developed a Web application. A user of the Web application selects the desired function and provides the parameter(s) for the selected function. The Web application then 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.

Our future plans include implementing a lambda reduction machine in XSLT, which will be an interpreter for the purely functional "lambda calculus" language. It will then be possible to specify the meaning of a Web service as a lambda calculus function (provided that the denotational specification of the language used to code the Web service is available), and send this function to the client, toghether with the lambda reduction machine, for execution on the client side. This can be useful in deciding whether the sevice provided meets the requirements of the client.

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. Ruhsan Onder and Zeki Bayram. XIM codes and the XIM interpreter in XSLT. Available at http://itrc.emu.edu.tr, 2006.
10. Michael Kay (Editor). XSL transformations (XSLT) version 2.0 of W3C working draft. Available at http://www.w3.org/TR/xslt20, 2005.
11. MSXML Documentation Team. Microsoft XML core services (MSXML) 1.0. Available at http://www.nedcomp.nl/support/origdocs/xml4/, 1998.
12. 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.
13. Jonathan Burns, Michael Lauzon, and Richard Anthony Hein. XPL:eXtensible programming language specification. Available at http://www.topxml.com/xpl/spec_draft.asp, 2000.
14. Fabio Arjona Arciniegas. *C++/XML*, chapter 13 (Creating XML-based Extension Languages for C++ Programs). New Riders Publishing, 2001.
15. 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.
16. W3C SMIL developer team. SMIL:synchronized multimedia integration language. Available at http://www.w3.org/AudioVideo/, 2006.
17. Nils Klarlund. ReaX: an undoable, timed synchronization language. Available at http://www.clairgrove.com/papers/ReaX-language-draft.html, 2000.
18. WML Programming Team. WML: Wireless markup language. Available at http://www.webreference.com/authoring/languages/xml/wml.html, 2002.
19. Linda Boyer, Peter Danielsen, and Jim Ferrans. VoiceXML: Voice extensible markup language version 1.0. Available at http://www.w3.org/TR/2000/NOTE-voicexml-20000505/, 2000.

20. Web3D Consortium. X3D. Available at http://xml.coverpages.org/ni2004-08-02-a.html, 2004.
21. Azzedine Boukerche, Diego Daniel Duarte, and Regina Borges de Araujo. VEML: A mark up language to describe web-based virtual environment through atomic simulations. In *Proceedings of the Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'04)*, pages 214 – 217, 2004.
22. David Carlisle and Robert Miner. MathML:mathematical markup language. Available at http://www.w3.org/Math/, 2006.
23. Elliotte Rusty Harold. Chapter 20 of the XML bible, second edition : XPointers. Available at http://www.cafeconleche.org/books/bible2/chapters/ch20.html, 2002.
24. OASIS development team. ebXML. Available at http://www.ebxml.org/, 2006.
25. Kimanzi Mati. Superx++. Available at http://xplusplus.sourceforge.net/indexPage.htm, 2006.
26. Barbara Carminati, Elena Ferrari, and Elisa Bertino. Securing XML data in third-party distribution systems. In *Proceedings of the 14th ACM International conference on Information and knowledge management (CIKM '05)*, pages 99–106, New York, NY, USA, 2005. ACM Press.
27. Tao-Ku Chang and Gwan-Hwan Hwang. Using the extension function of XSLT and DSL to secure XML documents. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA'04)*, pages 152 – 164, 2004.

APPENDIX

A 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 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>
```