INTERPRETING IMPERATIVE PROGRAMMING LANGUAGES IN EXTENSIBLE STYLESHEET LANGUAGE TRANSFORMATIONS (XSLT)

Ruhsan Onder
Department of Computer Engineering
Eastern Mediterranean University
Famagusta, Cyprus
email: ruhsan.onder@emu.edu.tr

Zeki Bayram
Department of Computer Engineering
Eastern Mediterranean University
Famagusta, Cyprus
email: zeki.bayram@emu.edu.tr

ABSTRACT
We use XSLT to implement an interpreter for a simple XML based imperative programming language called "XIM." Our work shows that not only is it theoretically possible to use XSLT as a programming language processor, but also that this is practically feasible. This has potential application in the area of delivering executable content over the Internet.

KEY WORDS
Executable Content, Imperative Languages, Interpreters, Operational Semantics, XML, XSLT

1 Introduction

Extensible Stylesheet Language Transformations (XSLT) was originally conceived as a way to transform data encoded as an XML document into an HTML page that can be viewed on a user's browser. The motivation behind representing data in XML format was to separate content from presentation of the content, with XSLT acting as the bridge between the two. XSLT works by transforming a source document through the application of templates. A template has two main parts: (1) a potentially complex pattern in the form of an XPATH expression which specifies a part of the source document, and (2) the way in which the specified part will be transformed.

In this paper we show that XSLT can be used to specify and implement the operational semantics of imperative programming languages. We define, using XML Schema, a simple XML based imperative programming language (XIM) and implement an interpreter for it in XSLT . 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. This approach opens the way for sending executable content over the Internet, together with the "executer" of the executable content. Programs written in a language can be sent to a Web browser, together with the language processor and data that the program requires, with obvious advantages.

Our literature search on XML, XSLT and programming language semantics leads us to believe that our approach of using XSLT for specifying the operational semantics of imperative programming languages is novel. In [1] a generic XML based scripting language and its interpreter which is implemented in C++ is explained. [2] introduces X-VRML, an XML based language for modeling virtual reality. The virtual scenes are dynamically generated from virtual scene models coded in X-VRML whose interpreter is written in Java. In [3] XPEN, an XML based format for distributed online handwriting recognition is explained. They use XSLT to translate XPEN into the Scalable Vector Graphics (SVG) format for visualisation and the processing of XPEN using a programming language via DOM. [4] introduces an incremental transformation framework for the editing of XML documents through one or many of the document's rendered presentations, in an interactive authoring system. To achieve this, extending transformation processors to be the basis of XML documents manipulation is proposed. In [5], the authors develop a lazy parser for the partial evaluation of input XML document trees in order to provide the pipelining of transformation sequences. They defer the actual parsing and document tree construction until the nodes are queried by the consumer. [6] introduces a statically typed XML processing language XDuce that provides constructors and destructors for XML documents which are regarded as node sequences and are primitive data values, where document schemas correspond to types. In [7], the authors take the same idea one step further and develop CDuce which is another XML processing language which implements overloaded operators, higher order functions, powerful sequence extracting patterns, records and tags as first-class expressions on top of the functionalities of XDuce.

The remainder of this paper is organized as follows. In section 2 we informally describe the syntax and semantics of XIM. Section 3 contains a brief description of an interpreter for XIM in XSLT. In section 4 the execution of the interpreter is traced on a simple XIM program. Finally, in section 5 we have the conclusion and future research directions.

2 The Minimal Imperative Language XIM

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).

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

  <main>
       <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>
```

Figure 1. Sample XIM program for computing 5!

var *fact* $\leftarrow$ 1
var *last* $\leftarrow$ 5
begin
  while (*last*>1) do
    fact $\leftarrow$ *fact*last*
    last $\leftarrow$ *last-1*
  end while
end

Figure 2. Pseudo-code of the sample XIM program

result_tree $\leftarrow$ XML document formed by the application of the *initializer* stylesheets to the original XIM program

While (the instruction to execute in *result_tree* is not the <*end*> element)
  result_tree $\leftarrow$ result of applying the *interpreter* stylesheet to *result_tree*
End While

Display the contents of the <*memory*> element

Figure 3. Pseudo-code of the top level *applier* program

```
<xsl:template match="prog/main">
    <xsl:variable name="current_inst"
  select="/prog/memory/var_declare[@name='PC']"/>

    <xsl:for-each
        select=".//child::*[@seq=$current_inst]">
        <!--there exists only one-->
        <xsl:call-template name="Execute">
            <xsl:with-param name="currentel"
                           select="."/>
        </xsl:call-template>
    </xsl:for-each>

    <xsl:element name="main">
         <xsl:for-each select="child::*">
              <xsl:copy-of select="."/>
         </xsl:for-each>
     </xsl:element>
</xsl:template>
```

Figure 4. Code fragment for identifying the element representing the current instruction and calling template `Execute` for executing it

A program in XIM is structured under a <`program`> root element, which in turn consists of the elements <`vars`> and <`main`>. The <`vars`> element encapsulates the variable declarations and the <`main`> element encapsulates the statements of the program. All XML tags representing XIM program segments have meaningful names which should be easily associated with well-known programming language features, and we shall not elaborate on the syntax and semantics of XIM any further. An example XIM program for computing 5! (five factorial) is shown in Figure 1 while its pseudo code is depicted in Figure 2.

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 that follows. 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 specify them explicitly- they are automatically generated, using XSLT, by the *initializer* stylesheets, described in the next section.

3 The Interpreter for XIM in XSLT

Three stylesheets are used to implement the operational semantics, together with a higher level imperative program, to apply the stylesheets. The higher level program is necessary because applying a stylesheet to an XML document is a one-time operation which results in a new XML document. What we need is the facility of applying a stylesheet repetitively to an XML document and the intermediate XML documents until we end up in an XML document which satisfies a termination condition, and the higher level code provides this.

Two *initializer* stylesheets are applied, one after the

```
1. <xsl:template name="Execute">
2.    <xsl:param name="currentel"/>

3.    <xsl:choose>
4.       <xsl:when test="name($currentel)='Assign'">
5.          <xsl:call-template name="assignment">
6.             <xsl:with-param name="c"
                            select="$currentel"/>
7.     <!--parameter holding the whole assign node-->
8.          </xsl:call-template>
9.       </xsl:when>

10.     <xsl:when test="name($currentel)='if' or
                    name($currentel)='while'">
11.        <xsl:call-template name="Construct">
12.           <xsl:with-param name="c"
                           select="$currentel"/>
13.        </xsl:call-template>
14.     </xsl:when>

15.     <xsl:when
      test="name($currentel)='statement_list'">
16.        <xsl:call-template name="Execute">
17.           <xsl:with-param name="currentel"
               select="$currentel/child::*[1]"/>
18.           <!-- first child of statement list-->
19.        </xsl:call-template>
20.     </xsl:when>

21.     <xsl:otherwise/> <!-- end -->
22.   </xsl:choose>
23.</xsl:template>
```

Figure 5. Code of template `Execute`

```
1.<xsl:template name="Assignment">
2.   <xsl:param name="c"/>
3.   <xsl:variable name="varname">
4.      <xsl:value-of select="$c/@varn"/>
5.   </xsl:variable>
6.   <xsl:element name="memory">
7.      <xsl:for-each
        select="/prog/memory/child::*[@name!='PC']">
8.         <xsl:choose>
9.            <xsl:when test="@name!=$varname">
10.              <xsl:copy-of select="."/>
11.           </xsl:when>
12.           <xsl:otherwise>
13.              <xsl:variable name="expr">
14.                 <xsl:call-template
                                  name="Evaluate">
15.                    <xsl:with-param name="n"
                          select="$c/child::*[1]"/>
16.                 </xsl:call-template>
17.<!--evaluate the op to get the value of expression-->
18.              </xsl:variable>

19.              <xsl:element name="var_declare">
20.                 <xsl:attribute name="name">
21.                    <xsl:value-of
                               select="$varname"/>
22.                 </xsl:attribute>
23.                 <xsl:value-of
                          select="number($expr)"/>
24.                   <!--new(assigned) value-->
25.              </xsl:element>
26.           </xsl:otherwise>
27.        </xsl:choose>
28.     </xsl:for-each>
29.     <xsl:element name="var_declare">
30.        <xsl:attribute name="name">
31.           <xsl:value-of select="'PC'"/>
32.        </xsl:attribute>
33.        <xsl:value-of select="$c/@next"/>
34.     </xsl:element>
35.  </xsl:element> <!--memory-->
36.</xsl:template>
```

Figure 6. Template `Assignment` to handle the assignment operation

other, to the XML document representing the XIM program to make it ready for interpretation. The first *initializer* stylesheet introduces the "program counter" and inserts sequence numbers into instructions as `@seq` attributes. The second *initializer* stylesheet then inserts the values in the `@next`, `@true_next` and `@false_next` attributes of the instructions. These attributes specify the address of the next instruction to execute. After the initialization stage, the *interpreter* stylesheet is applied to the result of these transformations, and is then repeatedly applied to the result of the previous application in a loop.

Figure 3 shows the abstracted pseudo-code version of the *applier* program performing the initialization and interpretation tasks. The actual implementation has been done using the Microsoft .NET framework and the C# language.

3.1 The *interpreter* stylesheet

The *interpreter* stylesheet carries out the execution of statements and evaluation of expressions using recursive templates. First it identifies the next instruction to execute and calls the template `Execute` to carry out the execution. The whole subtree representing the instruction is passed as an argument to the template `Execute`.

The instruction whose `@seq` attribute is equal to the value in the `PC` variable is the one to be executed next. As shown in Figure 4, the XSLT variable `$current_inst` is initialized to hold the value of the `PC` variable (line 2). The XPath expression in line 3 then identifies the subtree representing the instruction, and sends it "as a whole" to the named template `Execute` (lines 5-7).

3.1.1 The `Execute` template

The `Execute` template whose code is given in Figure 5, checks the type of the statement it is handed and if it is an `<assign>` element, calls the `Assignment` template (lines 4-9). For `<if>` and `<while>` constructs it calls the `Construct` template (lines 10-14) and for `<statemet_list>` elements it calls itself with the first instruction (first child) in `<statemet_list>` as a parameter (lines 15-20). For determining the type of the instruction, the XSLT `<choose>` construct is used (lines 3-22). Once the template that executes the instruction is done, control is returned back to the `Execute` template, which in turn returns control to its caller, where the `<main>` part of the XIM program is copied as-is to the new XML document (lines 9-13 of Figure 4). So the only changes that are made to the current version of the document at run-time are the contents of the `PC` variable, and possibly the values of other variables.

3.1.2 The `Assignment` template

The `Assignment` template, depicted in Figure 6, receives as its only argument the XML sub-tree consisting of an

<assign> statement in the `$c` parameter (line 2). The template also has access to the XML document as a whole.

It copies the name of the program variable to whom a new value is to be assigned into the XSLT variable `$varname` (lines 3-5) and recreates the whole <memory> element of the XML document (lines 6-35) to reflect the updated value of the variable (lines 19-25). It also updates the value of the *"program counter"* (lines 29-34). 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.

To determine the value to be assigned, the `Assign` template calls another template named `Evaluate` with the first and in fact the only child of the incoming <assign> element. The value returned as a result of this template call is assigned to XSLT variable `$expr` (lines 13-18).

3.1.3 Execution of the "If-then-else" and "While" constructs

The template `Construct` is depicted in Figure 7. The instruction which is an <if> or <while> construct comes into parameter `$c` (line 2). The truth value of the condition is determined and assigned to the XSLT variable `$condition` (lines 3-7). Then the whole <memory> element with all its children, except the `PC`, is copied (lines 9-11). 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 (lines 17-19). Otherwise the value of `@false_next` attribute is assigned to it (20-25). The main part of the program document is then copied as-is once control returns to the top-level template which matches `prog/main`.

3.1.4 Evaluation of Boolean and Arithmetic Expressions

The `Evaluate` template takes as a parameter an XML subtree representing an expression, evaluates it, calling itself recursively if necessary to evaluate the sub-expressions, and returns its result as a numeric or boolean value. Due to space restrictions its code is not shown here.

4 Execution Trace of the Interpreter on a Sample Program

In this section we show the execution trace of the interpreter on the sample XIM program given in Figure 1.

The top-level C# program initially applies the first *initializer* stylesheet which changes the root from <program> to <prog>, the <vars> child of the root to <memory>, inserts the variable `PC` into the set of user-defined variables, and assigns distinct values to the `@seq` attributes of instructions. The result of this transformation

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

3.   <xsl:variable name="condition">
4.      <xsl:call-template name="Evaluate">
5.         <xsl:with-param name="n"
                select="$c/condition/child::*[1]"/>
6.      </xsl:call-template>
7.   </xsl:variable>

8.   <xsl:element name="memory">
9.      <xsl:for-each
         select="/prog/memory/child::*[@name!='PC']">
10.        <xsl:copy-of select="."/>
           <!--copy all vars -->
11.     </xsl:for-each>
12.     <xsl:element name="var_declare">
13.        <xsl:attribute name="name">
14.           <xsl:value-of select="'PC'"/>
15.        </xsl:attribute>

16.        <xsl:choose>
17.           <xsl:when test="$condition='true'">
18.              <xsl:value-of
                   select="$c/condition/@true_next"/>
19.           </xsl:when>
20.           <xsl:otherwise>
                  <!-- condition=0 (false) -->
21.              <xsl:value-of
                  select="$c/condition/@false_next"/>
22.           </xsl:otherwise>
23.        </xsl:choose>
24.     </xsl:element>
25.  </xsl:element> <!--memory-->
26.</xsl:template>
```

Figure 7. Template `"Construct"` to handle "While" and "If-then-else" costructs

```
<prog>
  <memory>
    <var_declare name="fact"> 1 </var_declare>
    <var_declare name="last"> 5 </var_declare>
    <var_declare name="PC">1</var_declare>
  </memory>
  <main>
    <while seq="1">
      <condition>
        <boolop opname="gt">
          <var_use name="last" />
          <num> 1</num>
        </boolop>
      </condition>
      <statement_list seq="6">
        <assign varn="fact" seq="7">
          <op opname="*">
            <var_use name="fact" />
            <var_use name="last" />
          </op>
        </assign>
        <assign varn="last" seq="11">
          <op opname="-">
            <var_use name="last" />
            <num> 1</num>
          </op>
        </assign>
      </statement_list>
    </while>
    <end seq="15">
    </end>    <!-- program termination-->
  </main>
</prog>
```

Figure 8. Sample program after the application of the first *initializer* stylesheet

```
<prog>
  <memory>
    <var_declare name="fact"> 1 </var_declare>
    <var_declare name="last"> 5 </var_declare>
    <var_declare name="PC">1</var_declare>
  </memory>
  <main>
    <while seq="1" next="15">
      <condition true_next="6" false_next="15">
        <boolop opname="gt">
          <var_use name="last" />
          <num> 1</num>
        </boolop>
      </condition>
      <statement_list seq="6" next="1">
        <assign varn="fact" seq="7" next="11">
          <op opname="*">
            <var_use name="fact" />
            <var_use name="last" />
          </op>
        </assign>
        <assign varn="last" seq="11" next="1">
          <op opname="-">
            <var_use name="last" />
            <num> 1</num>
          </op>
        </assign>
      </statement_list>
    </while>
    <end seq="15">
    </end>    <!-- program termination-->
  </main>
</prog>
```

Figure 9. Sample program after the application of the second *initializer* stylesheet

is given in Figure 8. Then the second *initializer* stylesheet is applied to the result of the first transformation and the missing `@next` attributes are filled in by this transformation, as shown in Figure 9. This is followed by the repeated application of the *interpreter* stylesheet. Figures 10-22 depict the changes in the values of the variables in the <`memory`> element after each application of the *interpreter* stylesheet. Note the change in the value of `PC` which holds the sequence of instruction to be executed next.

At the first application of the *interpreter,* the value of `PC` becomes 6 since the condition of <`while`> is satisfied (Figure 10). Next, the first assignment instruction in the <`statementlist`> (having `@seq`=6) is executed and value of `$fact` becomes 5 while `PC` becomes 11 (which is the value of `@seq` attribute of the next instruction to be executed) (Figure 11). When the assignment statement which has `@seq`=11 is executed, the value of `$last` is decremented to 4 and `PC` becomes 1, to provide one more iteration of the loop (Figure 12). At each application of the *interpreter* the value of `$last` is multiplied with `$fact` and then decremented, until the value of `$last` becomes 1 (through Figures 13- 21). When the value of `$last` becomes 1, the condition of <`while`> becomes false and `PC` is updated to 15 (which is the value of `@seq` attribute of the <`end`> element) to reflect the end of program (Figure 22). Then, the higher level *applier* C# program detects the program termination and stops, because before each application of the *interpreter* it checks whether the instruction to be executed is an <`end`> statement.

```
<var_declare name="fact"> 1 </var_declare>
<var_declare name="last"> 5 </var_declare>
<var_declare name="PC"> 6 </var_declare>
```

Figure 10. Sample program after the execution of the first assignment statement in the loop

```
<var_declare name="fact">5</var_declare>
<var_declare name="last"> 5 </var_declare>
<var_declare name="PC">11</var_declare>
```

Figure 11. Sample program after the execution of the second assignment statement in the loop

```
<var_declare name="fact">5</var_declare>
<var_declare name="last">4</var_declare>
<var_declare name="PC">1</var_declare>
```

Figure 12. Memory part of the sample program after `$last` decremented and `PC` is updated as 1 to loop

```
<var_declare name="fact">5</var_declare>
<var_declare name="last">4</var_declare>
<var_declare name="PC">6</var_declare>
```

Figure 13. Condition of <`while`> is satisfied and `PC` gets value of `@true_next`

```
<var_declare name="fact">20</var_declare>
<var_declare name="last">4</var_declare>
<var_declare name="PC">11</var_declare>
```

Figure 14. codefrag$fact = $fact * $last

```
<var_declare name="fact">20</var_declare>
<var_declare name="last">3</var_declare>
<var_declare name="PC">1</var_declare>
```

Figure 15. `$last` is decremented and `PC` becomes 1 to loop

```
<var_declare name="fact">20</var_declare>
<var_declare name="last">3</var_declare>
<var_declare name="PC">6</var_declare>
```

Figure 16. Condition of <`while`> is satisfied and `PC` gets value of `@true_next`

```
<var_declare name="fact">60</var_declare>
<var_declare name="last">3</var_declare>
<var_declare name="PC">11</var_declare>
```

Figure 17. `$fact = $fact * $last`

```
<var_declare name="fact">60</var_declare>
<var_declare name="last">2</var_declare>
<var_declare name="PC">1</var_declare>
```

Figure 18. `$last` is decremented and `PC` becomes 1 to loop

```
<var_declare name="fact">60</var_declare>
<var_declare name="last">2</var_declare>
<var_declare name="PC">6</var_declare>
```

Figure 19. Condition of <`while`> is satisfied and `PC` gets value of `@true_next`

```
<var_declare name="fact">120</var_declare>
<var_declare name="last">2</var_declare>
<var_declare name="PC">11</var_declare>
```

Figure 20. `$fact = $fact * $last`

```
<var_declare name="fact">120</var_declare>
<var_declare name="last">1</var_declare>
<var_declare name="PC">1</var_declare>
```

Figure 21. `$last` is decremented and `PC` becomes 1 to loop

```
<var_declare name="fact">120</var_declare>
<var_declare name="last">1</var_declare>
<var_declare name="PC">15</var_declare>
```

Figure 22. Sample program after the *applier* program terminates. Condition of <`while`> evaluates to false and `PC` gets value of `@false_next` to terminate loop

5 Conclusion and Future Research Directions

We defined an XML based imperative language called "XIM" using XML Schema and implemented the operational semantics of XIM in XSLT. The implementation consists of three XSLT stylesheets, two of which are applied only one time each to make the source program ready for interpretation. The third stylesheet does the actual interpretation and is applied repeatedly until the program terminates (provided that it does).

Our work shows that XML Schema and XSLT, together with some facility for repeated application of XSLT stylesheets, are sufficient to describe fully the syntax and operational semantics of XML-based imperative programming languages. Although XIM is minimal, with only the most basic programming constructs, ideas similar to the ones that are used for the implementation of its operational semantics can be used for more complex languages.

We believe this ability of using XSLT to implement programming language semantics can have a profound impact on the delivery of executable content on the Internet. With appropriate enhancements to the XSLT processors of Web browsers (such as specifying the order and number of times in which stylesheets will be applied) it will be possible to send (i) data, (ii) a program to manipulate the data, and (iii) the interpreter to run the program, all at the same time, to a browser, which will in turn execute the program using the interpreter.

We intend to take this line of research further by investigating the possibility of *compiling* XIM into an intermediate abstract machine code and interpreting the resulting machine code, all using XSLT. We then intend to use XSLT to implement the denotational semantics of XIM by translating XIM programs into lambda calculus, and interpreting the lambda calculus code.

References

[1] F. A. Arciniegas. Creating C++ interpreters for XML extension languages. *http://www.informit.com/articles/article.asp?p=23277*, September 2001.

[2] K. Walczak and W. Cellary. X-VRML - XML based modeling of virtual reality. *Proceedings of the 2002 symposium on applications and the internet (SAINT'02)*, pages 204–213, 2002.

[3] K.P. Lenaghan and R.R. Malyan. XPEN: An XML based format for distributed online handwriting recognition. *Proceedings of the seventh international conference on document analysis and recognition (ICDAR'03)*, pages 1270–1274, 2003.

[4] L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. *Proceedings of the eleventh international conference on World Wide Web*, pages 474–485, 2002.

[5] S. Schott and M. L. Noga. Lazy XSL transformations. *Proceedings of the 2003 ACM symposium on Document engineering*, pages 9–18, 2003.

[6] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[7] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63, 2003.