Acta Cybernetica **15** (2002) 509–528.

XML Semantics Extension

Ferenc Havasi*

Abstract

Nowadays one of the best standards for storing structured data is XML. The key idea behind our paper is based on a relationship between XML documents and attribute grammars. This parallel makes it possible to apply techniques of attribute grammars (semantics rules) to the XML environment.

We present a new method for extending XML. After a background discussion we formally introduce a way of defining real XML attributes using semantics rules. This allows us to incorporate semantics rules into XML files in such a way that it does not violate the original XML specification and is suitable for compressing using learning. After learning an associated semantics rule file preserves the relationship between attributes.

Introduction

These days one of the most popular standards in use for storing structured information is XML. More and more applications are able to export data in an XML format, more databases are stored in XML, and XML processing techniques are becoming more generic. If this trend continues, XML will eventually be present in almost every part of the informatics sphere. Because of this the new research results related to XML should prove important the future.

The main idea behind our paper is based on a connection between XML documents and attribute grammars. The analogy makes it possible to apply techniques of attribute grammar (semantics rules) to the XML environment. The first notion of including semantics to XML was published in [9], but here we shall introduce a new approach. We create a format which makes it possible for us to define a real XML attribute via semantics rules. The new set of semantics rules then become an organic part of XML documents and do not violate the original XML specification.

In the first section we review the fundamentals of XML and attribute grammars. Next, we throw light on the relationship between them, show how to extend XML documents with semantics rules, and then introduce a new (SRML) format to describe it. In Section 3 we discuss new S-SRML and L-SRML descriptions, which are counterparts of S-attributed and L-attributed grammars. The next section deals with the learning of SRML descriptions. In Section 5 we discuss the

*Research Group of Artificial Intelligence, Hungarian Academy of Sciences, Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544145, email: havasi@rgai.hu

implementation, and present experimental results. Finally we briefly elaborate on related and future work, and give a summary of our conclusions.

1 Background

To facilitate an understanding of this paper we first give a brief overview of the necessary fundamentals such as XML technology, formal and attribute grammars. Rather than give a detailed account of the above, we will just present the most important parts using an example.

1.1 XML

In this section we provide a brief introduction to the basics of XML [2], which we will make use of later.

1.1.1 Document:

An XML document has an html like text-based format. Its components are called *elements*. An *element* always begins with a *start-tag* and ends with an *end-tag*. Take, for instance,

```
<section>A long text</section>
```

An element may contain other elements and/or text or it may be empty. With the start-tag of an element it is possible to define *attributes*, for example,

```
<section title="Introduction">A long text</section>
```

In Figure 1 we have a more complicated example using a numeric expression.

1.1.2 DTD:

This is a file containing a meta language. Its description makes it possible for us to define the structure of an XML document. This language is called *DTD*. We can specify the content of an element, which other element or text can be inserted, and in which order. In our case the following two meta-tags are the most important:

!element: This tag specifies a regular expression[1]. It defines the element and the order it assumes.

!attlist: This tag specifies the type and allowed values of the attributes of an element.

Figure 2 shows one such DTD file:

1. The element num can contain only text (#PCDATA), and has a required (#REQUIRED) attribute called type, its value being real or integer-valued.

[1]It is possible to transform this expression to EBNF format [9].

```
<expr>
  <multexpr op="mul" type="real">
    <expr type="int">
      <num type="int">3</num>
    </expr>
    <expr type="real">
      <addexpr op="add" type="real">
        <expr type="real">
          <num type="real">2.5</num>
         </expr>
         <expr type="int">
            <num type="int">4</num>
         </expr>
       </addexpr>
    </expr>
  </multexpr>
</expr>
```

Figure 1: A possible XML form of the expression 3*(2.5+4).

```
<!ELEMENT num (#PCDATA) >
<!ATTLIST num type ( real | int ) #REQUIRED
>
<!ELEMENT expr ( num |  multexpr  |  addexpr ) >
<!ATTLIST expr type ( real | int ) #IMPLIED
>
<!ELEMENT multexpr ( expr , expr ) >
<!ATTLIST multexpr op  ( mul | div )  #REQUIRED
                   type ( real | int ) #IMPLIED
>
<!ELEMENT addexpr ( expr , expr ) >
<!ATTLIST addexpr op  ( add | sub )  #REQUIRED
                  type ( real | int ) #IMPLIED
>
```

Figure 2: The DTD of the corresponding XML file for the previous figure.

2. The expr element contains a num or a multexpr or an addexpr element (| mark means or), and there is an optional (#IMPLIED) attribute called type.

3. The elements multexpr and addexpr must contain two expr elements, and have two attributes: a necessary one called op and an optional one called type.

1.2 Parsing XML

Basically there are two kinds of interfaces of XML parsers:

SAX interface: This interface is very simple: all information is passed from the parser to the application via function calls. For example if there is a *start-tag* the parser generates a startTag(element_name)-like call.

DOM tree interface: Using this method the parser builds a tree called a DOM tree. Every element and attribute is represented as a vertex. The root node of the tree is the root element of the document. The edges represent incorporated dependences. A corresponding tree of the example in Figure 1 is shown in Figure 3.

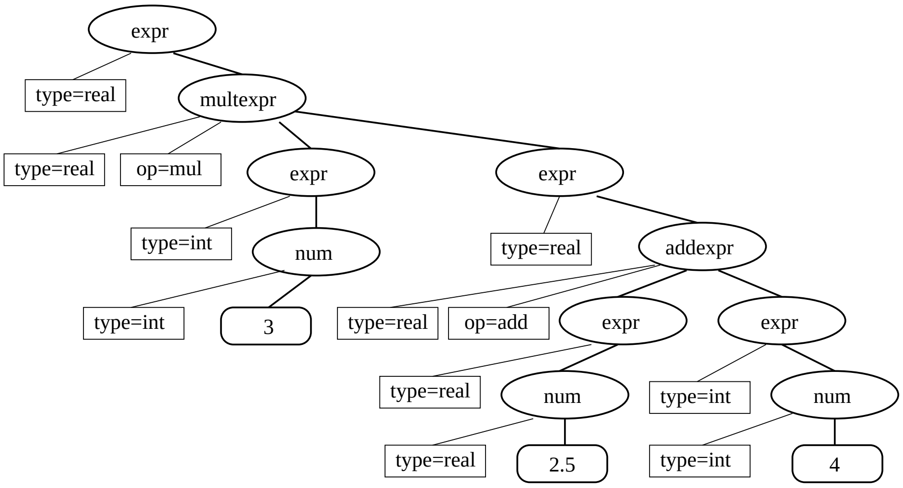


Figure 3: DOM tree of example in Figure 1.

1.3 Formal languages and derivation trees

A general way of specifying a syntax is by using a formal grammar. A formal grammar is a G=(N,T,S,P) set, where N is the set of the nonterminal symbols, T is the one of the terminal symbols, S is the start-symbol and P is a set of transformation rules. Take, for example,

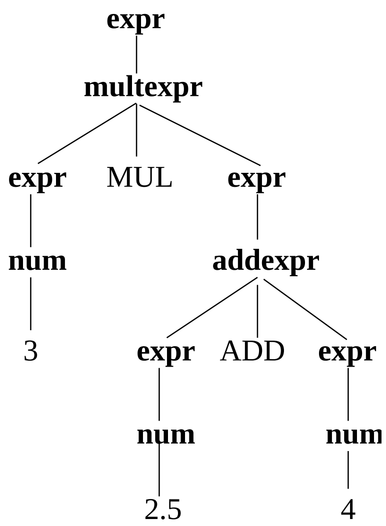
```
N = { expr, multexpr, addexpr, num }
S = expr
T = { "ADD" , "MUL" , NUM }
P :
  expr     -> num | multexpr | addexpr
  addexpr  -> expr "ADD" expr | expr "SUB" expr
```

```
multexpr -> expr "MUL" expr | expr "DIV" expr
num      -> NUM
```

The grammar in it specifies the format of simple numeric expressions. At the left side of every rule there is only one nonterminal, so it is a context free (CF) grammar. The derivation process starts at *expr*. It may be a simple number (*num*), a multiplicative or an additive expression (*multexpr* or *addexpr*). The *num* nonterminal is a simple number token (NUM), whereas the multiplicative expression contains two expressions and a MUL or DIV token is placed between them. The additive expression also consists of two expressions with an ADD or SUB token between them.

For any given input word a derivation tree can be drawn. In the root of a derivation tree there is the start-symbol (expr), and below nonterminals there are substituted expressions. If we concatenate the leaves we get the input word.

If this input is 3 MUL (2.5 ADD 4), the derivation tree looks like the following:



1.4 Attribute grammars

An attribute grammar [7] contains a CF grammar, and

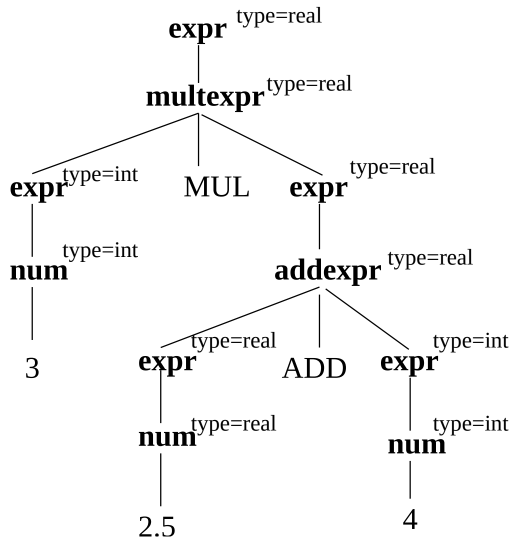
attributes: We can assign attributes to *nonterminals*. For example, by assigning the x attribute to the S nonterminal we get the S.x attribute. In general there are two types of attributes: inherited and synthesized.

semantics rules: We can assign semantics rules for each *formal rule*. The semantics rules define a formula for computing the value of an attribute. For example:

```
S -> A B
  S.x = A.x + B.x
```

In this semantics rule the attribute x of S can be calculated from the attribute x of A and B.

If we supplement the derivation tree with the values of attribute occurrences we get an attributed derivation tree. All attribute occurrences are calculated once and only once. Take, for instance the attributed derivation tree of the expression example:



2 XML semantics

2.1 Relationship between attribute grammars and XML

Figure 3 shows the DOM tree of the previous XML document example. If we recall the attributed derivation tree, we will recognize that an XML document can be viewed as an attributed derivation tree. Hence we expect to see the following analogy:

Attribute Grammars	XML
nonterminal	element
formal rules	element specification in DTD
attribute specification	attribute specification in DTD
semantic functions	???

There is one key concept in AG that has no counterpart in the XML environment: semantic functions. This is a very important concept in AG and there are a lot of techniques based on it. It might be useful to apply these techniques to the XML environment.

In an XML document the values of all attribute occurrences are stored in a direct form. If we were able to define semantics rules for XML attributes it would be sufficient to store these rules once and, making use of them, the concrete value of the attributes could be calculated. Then it would not be necessary to store them in the document.

So our idea is to define XML attributes using semantics rules. Our definition of semantics rules will be an organic part of XML document.

2.2 Complete/Reduce

We will define only IMPLIED XML attributes with semantics rules. In an XML document this kind of attribute is not required, so the DTD of the document validates the XML document whether or not these attributes are defined.

Now let us consider the following figure:

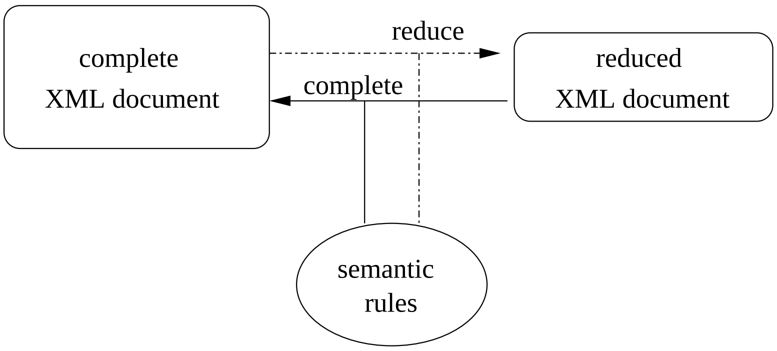


Figure 4: The complete and reduce methods.

In this diagram we notice that the *complete* method creates a complete XML document from a reduced one. All nondefined IMPLIED attributes that have a semantics rule will be calculated using this rule.

In contrast, the *reduce* method does the opposite of the previous one. The input of it is a complete XML document, along with some semantics rules. All IMPLIED attributes which have a *correct*[2] semantics rule will be deleted from the document, so the output will be a reduced version of the input XML document.

2.3 Specifying semantics rules

We define a meta language called SRML (Semantics Rule Meta Language) to describe semantics rules, which has an XML based format. The corresponding DTD of this language is the following:

[2]Note that we can use rules even if they are not absolutely correct rules. If the result of the semantics rule differs from the actual attribute value it will be kept. So the *reduce* method is really the inverse of the *complete* method.

```
<!ELEMENT semantic-rules ( rules-for* ) >
<!ELEMENT rules-for ( rule* ) >
<!ATTLIST rules-for root  NMTOKEN  #REQUIRED>
<!ELEMENT rule ( expr ) >
<!ATTLIST rule element  NMTOKEN  #REQUIRED
               attrib  NMTOKEN  #REQUIRED
>
<!ELEMENT expr ( binary-op | attribute | data |
                 no-data | if-element | if-expr |
                 if-all | if-any | current-attribute |
                 position | external-function ) >
<!ELEMENT binary-op ( expr, expr) >
<!ATTLIST binary-op op  (add | sub | mul | div | exp | equal |
                         not-equal | less | greater | or |
                         xor | and | nor | contains | concat |
                         begins-with | ends-with )    #REQUIRED
>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute element NMTOKEN                   "srml:this"
                    num     NMTOKEN                   "0"
                    from    ( begin | current | end ) "current"
                    attrib  NMTOKEN                   #REQUIRED
>
<!ELEMENT if-element ( expr, expr)>
<!ATTLIST if-element from ( begin | end ) "begin">
<!ELEMENT position EMPTY>
<!ATTLIST position element  NMTOKEN                   "srml:all"
                   from     ( begin | end )           "begin"
>
<!ELEMENT if-all ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-all element    NMTOKEN                   "srml:all"
                 attrib     NMTOKEN                   "srml:all"
>
<!ELEMENT if-any ( expr, expr, expr)> <!-- cond, if, else -->
<!ATTLIST if-any element    NMTOKEN                   "srml:all"
                 attrib     NMTOKEN                   "srml:all"
>
<!ELEMENT current-attribute EMPTY>
<!ELEMENT if-expr (expr , expr , expr ) >
                                   <!-- condition , if, else -->
<!ELEMENT data (#PCDATA) >
<!ELEMENT no-data EMPTY>
<!ELEMENT extern-function (param)*>
<!ATTLIST extern-function name NMTOKEN #REQUIRED>
<!ELEMENT param (expr)>
```

A DTD can be viewed as a formal grammar [9]: elements will be nonterminals and the descriptions of the element will be formal rules. We would like to define some semantics rules for these formal rules.

The meaning of the elements of SRML are:

semantics-rules : This is the root element of SRML.

rules-for : This element gathers together the semantics rules of a formal rule. In a DTD there is only one working description of an element so the formal rule is determined by that element, which is on the left side of the formal rule. This is the *root* attribute of the *rules-for* element.

rule : This element describes a semantics rule. We have to specify which *attrib*ute of which *element* we are going to define in this semantics rule and its value in *expr*. If the value of the *element* is "*srml:root*" then we define an attribute of the root element.

expr : An expression can be a binary-expression (*binary-op*), an *attribute*, a directly defined value (*data* or *no-data*), a conditional expression (*if-expr*,*if-all* or *if-any*), a syntax-condition (*if-element* and *position*) or an external function call (*extern-function*).

if-element : In a DTD element description one can specify a regular expression (with maybe +,*,?,... marks). This element provides us with the possibility of testing the actual form of the input. It contains two *expr* elements. As an expression the value of the *if-element* is true or false depending on the following: the name of the first *expr*th child (element) in the actual rule equals to the value of the second *expr*. The *from* attribute can specify which direction to operate. In other words it is possible to take an index from the end of the level without actually knowing how many children the parent has.

binary-op : This element is an ordinary binary expression.

position : Returns a 0 based index which identifies the current element's position taking into consideration the *element* attribute. Possible directions are as follows : begin, end. It is possible to use the *srml:all* identifier, in which case the index returned will be the actual overall position in the DOM tree level. If an *element* name is specified then the returned index will be n where the element has $n-1$ predecessors or successors with the same element name.

attribute : The attribute is determined by its *element*, *attrib*, *from* and *num* attributes. In the actual rule this is the *num*th element where the name matches the value of the *element* (if it is "*srml:any*" it can be anything, if it is "*srml:root*" then it is an attribute of the root) *from* the *begin*ning, the *current* element or the *end*ing. If the attribute does not exist it will be handled as *no-data*.

if-expr : This is an ordinary conditional expression. The first *expr*ession is the condition and depends on whether if it is true (not zero) or not. The value of the *if-expr* will be the value of the second or third *expr*ession.

if-all : This is an iterated version of the previous *if-expr* expression. The first *expr* is computed for all matching attributes (each selected by *element* and *attrib*ute, which can take a concrete value or "*srml:all*)". We can refer to the value of this attribute using the element *current-attribute.* If the condition (first *expr*) is true for all matching attributes, the value of it is the value of the second *expr* otherwise it is the third *expr.*

if-any : This is almost the same as the previous one except that it is sufficient that the condition be true for at least one matching attribute.

current-attribute : This is the loop variable of *if-any* and *if-all* elements.

data : This element has no attribute and usually contains a number or a string.

no-data : This element means that this attribute cannot be computed – it is often present in some branches of conditional expressions.

extern-function : This element makes an external function call handled by the implementation. It makes SRML more easily extendable.

param : This describes a parameter of an *external-function.*

A valid SRML description must be *consistent.* This means there mustn't be any attribute occurrences that are defined more than once.

2.4 An SRML example

Here are some interesting portions of the SRML description from the previous numeric expression example.

```
<semantic-rules>
<rules-for root="expr">
<rule element="srml:root" attrib="type">
  <expr>
    <attribute element="srml:any" num=1 attrib="type" from="begin"/>
  </expr>
</rule>
</rules-for>
<rules-for root="addexpr">
<rule element="srml:root" attrib="type">
  <expr>
    <if-expr>
    <expr>
      <binary-op op=or>
```

```
        <expr>
        <binary-op op=equal>
          <expr>
            <attribute element="expr" num=1 attrib="type"
                                            from="begin"/>
          </expr>
          <expr><data>real</data></expr>
        </binary-op>
        </expr>
        <expr>
        <binary-op op=equal>
          <expr>
            <attribute element="expr" num=2 attrib="type"
                                            from="begin"/>
          </expr>
        <expr><data>real</data></expr>
        </binary-op>
        </expr>
      </binary-op>
    </expr>
    <expr><data>real</data></expr>
    <expr><data>int</data></expr>
    </if-expr>
  </expr>
</rule>
</rules-for>
</semantic-rules>
```

The set of rules for *multexpr* is very similar to that for *addexpr*.

3 Attribute grammar types – SRML description types

The attribute grammars can be classified according to the evaluation method employed [1]. There are S-attributed grammars, L-attributed grammars, OAG attributed [6] grammars and so on.

By analogy we could introduce S-, L-, ASE-,... SRML descriptions. Here we only define S- and L-SRML descriptions.

Actually, there are only two relevant factors which we need to know in the SRML description to decide whether it is an S/L-SRML description. The first one is the set of defined attributes associated with the rules, while the second is the set of referenced attributes in these definitions.

3.1 The S-SRML description

S attributed grammars are the simplest attribute grammars: they have only synthesized attributes. As in XML, in SRML we do not distinguish between synthesized attributes and inherited ones. In this environment we can define an S-SRML description, in analogy with S-attributed grammars:

definable attributes : In each rule we can only define the attributes of the (srml:)root nonterminal (the element which is on the left side), because synthesized attributes are only definable in a rule if the root element contains them.

usable attributes in definitions : All attributes in this rule presume that there are no circular dependencies.

3.2 The L-SRML description

An L-attributed grammar can contain synthesized and inherited attributes, but the dependencies between them must be evaluated in one left-to-right pass. In the SRML environment it means the following:

definable attributes : All available attribute occurrences bearing consistency (see in 2.3) in mind.

usable attributes in definitions : We use one left-to-right pass to evaluate the attributes. In a rule environment we first calculate the attributes of the children nonterminals, and after the attributes of the root nonterminal in a suitable order. An SRML description is called an L-SRML description if there is a suitable order in attributes which carries out the following: if there is an attribute reference in the definition of an attribute then the value of referenced attribute has already been calculated.
To be more precise:

- In the definition of an attribute of the root there can be attributes of any children, or those attributes of the root that have been defined earlier in the SRML description.
- In the definition of an attribute of a child there can be attributes of any children which lie to the left side of it, or are those attributes of the same child that have been defined earlier in the SRML description.

4 Learning of the SRML description

A relatively large XML document usually contains lots of redundancies. This means that many attributes can be computed from other attributes. However, in many cases the computation rules are not trivial and the recognition of these may require machine learning approaches [8]. The learning of the SRML description means detecting relationships between XML attributes. These relationships can be described in an SRML format. This method has two obvious applications, namely:

compressing: After learning an SRML description we can then the apply the *reduce* algorithm. This reduced version of the document is usually much smaller than the original one. From it and the SRML description the original document is recoverable, so it can be regarded as a form of data compression[3].

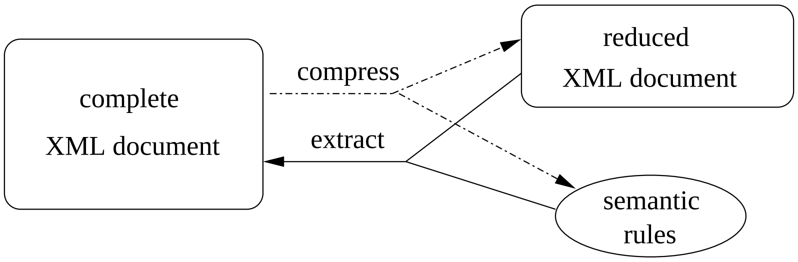


Figure 5: Using the learning of an SRML description to (de)compress XML files.

understanding: The SRML description provides us with hitherto unknown correspondences. When we store a database in XML document format we can find correspondences in it as well.

The learning method can be fault-tolerant. In this case the detected correspondences may be better (the data may have measuring errors). If we only need estimated values and the (XML) database is very large, we can use the short SRML description.

In [5] and [12] a machine learning method was introduced to infer the semantic rules of attribute grammars. There is a close relationship between attribute grammars and SRML, hence this learning approach can be used to produce SRML description from XML documents.

5 Implementation

The structure of the implementation is given in Figure 6. The implementation of a complete version of this tool is under way (in a JAVA and DOM environment). Here we present results using a special version of this tool. This tool's task is to read CPPML files and reduce them.

[3]Of course after this method we can use other ordinary compression methods like those mentioned in [10].

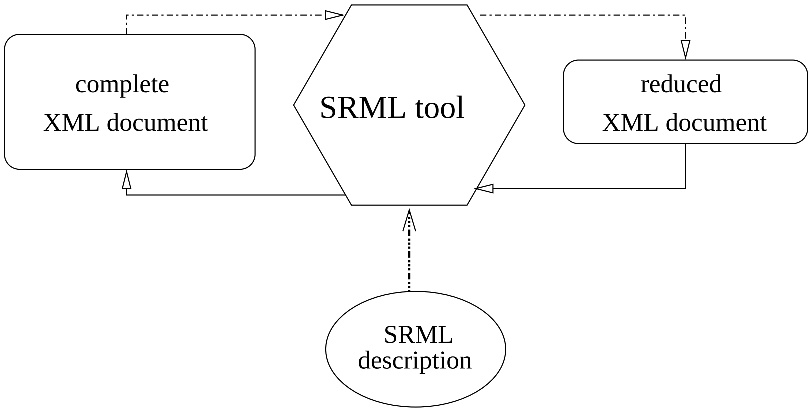


Figure 6: The structure of the implementation.

5.1 CPPML

CPPML (C++ Markup Language) [3] is a markup language for describing the structure of programs written in C++. This may be generated using a Columbus reverse engineering tool [4] for any C++ program.

Here is a brief extract of a C++ program:

```
class _guard : public std::map<std::string, _guard_info>
  {
    public:
        void registerConstruction(const type_info & ti)
    { (*this)[ti.name()]++ ;
    }

    void registerDestruction(const type_info & ti)
    { (*this)[ti.name()]-- ;
    }

    void dumpInstances(const char * file, bool bAppend)
    {
      fstream f(file, bAppend  ?  ios_base::out|ios_base::app
                               :  ios_base::out) ;
      iterator i ;
      for(i=begin(); i!=end(); i++)
        f  << i->second._count << "  -  " << i->second._max_count
           << "  :  "  << i->first<< endl ;
      f << endl;
    }
  } ;
```

```
} ;
```

The CPPML representation of the above could be like the following:

```
...
<class id="id20097" name="_guard"
    path="D:\CAN_Test\SymbolTable\Input\CANGuard.h" line="71"
    end-line="90" visibility="global" abstract="no"
    defined="yes" template="no" template-instance="no"
    class-type="class">
  <function id="id20102" name="registerConstruction"
      path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      line="75" end-line="76" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      body-line="75" body-end-line="76">
    <return-type>void</return-type>
    <parameter id="id20106" name="ti"
         path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
         line="74" end-line="74" const="yes">
      <type>type_info&amp;</type>
    </parameter>
  </function>
  <function id="id20109" name="registerDestruction"
      path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      line="79" end-line="80" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
      body-line="79" body-end-line="80">
    <return-type>void</return-type>
    <parameter id="id20113" name="ti"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="78" end-line="78" const="yes">
      <type>type_info&amp;</type>
    </parameter>
   </function>
   <function id="id20116" name="dumpInstances"
       path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
       line="83" end-line="89" visibility="public" const="no"
       virtual="no" pure-virtual="no" kind="normal"
       body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
       body-line="83" body-end-line="89">
     <return-type>void</return-type>
     <parameter id="id20120" name="file"
         path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
         line="82" end-line="82" const="yes">
```

```
      <type>char*</type>
    </parameter>
    <parameter id="id20122" name="bAppend"
        path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
        line="82" end-line="82" const="no">
      <type>bool</type>
    </parameter>
  </function>
</class>
...
```

5.2 A real example: reducing of CPPML

As you might have noticed in a CPPML description many attributes can be calculated or estimated using other attributes.

For example, the *kind* of a *function* is a *constructor* if the *name* of the *function* is equal to the name of the *class*. We can describe it in SRML form:

```
<rules-for root="class">
  <rule element="function" attrib="kind">
    <expr>
      <if-expr>
        <expr>
          <binary-op op="equal">
            <expr>
              <attribute attrib="name"/>
            </expr>
            <expr>
              <attribute attrib="name" element="srml:root"/>
            </expr>
          </binary-op>
        </expr>
        <expr>
          <data>constructor</data>
        </expr>
        <expr>
          ...
        </expr>
      </if-expr>
    </expr>
  </rule>
</rules-for>
```

Actually we can not only describe valid rules here, but also make estimations (the *reduce* method will only delete the matching attributes). Let us look at some types of estimations:

1. the declaration of a function or variable starts and ends on the same line.
2. the implementation of the functions of a class are usually in the same file.
3. the parameters of a function are also in the same file, perhaps on the same line.

Here are some parts of the corresponding SRML description:

```
<rules-for root="function">
  <rule element="parameter" attrib="end-line">
    <expr>
      <attribute attrib="line"/>
    </expr>
  </rule>
  <rule element="parameter" attrib="line">
    <expr>
      <attribute attrib="line" num="-1"/>
    </expr>
  </rule>
  <rule element="parameter" attrib="path">
    <expr>
      <attribute attrib="path" num="-1"/>
    </expr>
  </rule>
</rules-for>
```

The full description of our SRML description for CPPML can be found at the following internet site: `http://xml.rgai.hu/`.
Let us illustrate the previous part of the CPPML description using the procedure *reduce*:

```
<class id="id20097" name="_guard"
    path="D:\CAN_Test\SymbolTable\Input\CANGuard.h"
    line="71" end-line="90" visibility="global" abstract="no"
    defined="yes" template="no" template-instance="no"
    class-type="class">
  <function id="id20102" name="registerConstruction"
      line="75" end-line="76" visibility="public" const="no"
      virtual="no" pure-virtual="no" kind="normal"
      body-path="D:\CAN_Test\SymbolTable\Input\CANGuard.h">
    <return-type>void</return-type>
    <parameter id="id20106" name="ti" line="74" const="yes"
         ellipsis="no">
      <type>type_info&amp;</type>
    </parameter>
  </function>
```

```
  <function id="id20109" name="registerDestruction" line="79"
      end-line="80" const="no" virtual="no" pure-virtual="no"
      kind="normal" visibility="global">
    <return-type>void</return-type>
    <parameter id="id20113" name="ti" line="78" const="yes"
        ellipsis="no">
      <type>type_info&amp;</type>
    </parameter>
  </function>
  <function id="id20116" name="dumpInstances" line="83"
      end-line="89" const="no" virtual="no" pure-virtual="no"
      kind="normal" visibility="global">
    <return-type>void</return-type>
    <parameter id="id20120" name="file" line="82" const="yes"
         ellipsis="no">
      <type>char*</type>
    </parameter>
    <parameter id="id20122" name="bAppend" const="no"
       ellipsis="no">
      <type>bool</type>
    </parameter>
  </function>
</class>
```

5.3 Experimental results

The results of applying the *reduce* method are listed in Figure 7. The column "Orig Size" is the size of the original CPPML file, and "New Size" is its size after the reduction. The original CPPML file contained "Attribute" attribute items, and we eliminated "Deleted" ones because it was possible to compute their values using the rules. In the "Not matched" column there is the attribute number which is computable from the rules but its value is not correct. Here we do not delete these attributes.

Prog Name	Orig Size	New Size	Attributes	Deleted	Not matched
AppWiz	3589076	2192377	115684	43451	10151
jikes	2257728	1745720	93045	26100	9964
leda	11673855	9023687	405916	88322	26032

Figure 7: Results of CPPML reduction.

The size of this CPPML SRML description is less than 4 kilobytes, and from the reduced version of the document and this SRML description the original one can be recovered, so the *reduce* method is a form of data compression. After this "compression" we can apply some ordinary compression technique. The density of information in the *reduced* document is higher than in the original document, so it would be interesting to examine the compression ratio using a zip-like compression.

We compressed the original and the *reduced* document using gzip, and the results are shown in Figure 8. The difference between the compression ratio using gzip on the original and the *reduced* documents is about 1%.

Prog Name	Orig Size	Gziped orig	Orig ratio	New Size	Gzipped new	New ratio
AppWiz	3589076	244659	0.068	2192377	167749	0.076
jikes	2257728	177223	0.078	1745720	140681	0.081
leda	11673855	821074	0.070	9023687	709922	0.079

Figure 8: Results of compressing the original and the reduced CPPML description.

6 Note on a related study

The first notion of adding semantics to XML was published in [9]. After a brief introduction to XML that paper provides a method for transforming the element description of DTD into EBNF formal rule description.

Afterwards it introduces its own SRD (Semantics Rule Definition) composed of two parts: the first one describes the semantics attributes[4], while the second one gives a description of how to compute them. SRD is also XML-based.

The main difference between the approach outlined in this article and ours is that we provide semantics rules not just for newly defined attributes but also for real XML ones. Our approach makes the SRML description an organic part of XML documents. As the defined attributes are IMPLIED, either the complete or the reduced document is validated by the original DTD. This kind of semantics definition could offer a useful extension for XML techniques.

Our SRML description also differs from the SRD description in that article. In SRD the attribute definition of elements with a + or * sign is defined in a different way from the ordinary attributes definition and can only reference the attributes of the previous and subsequent element. The references in our SRML description are more generic, and all expressions are XML-based.

7 Conclusion and Future Work

Our method defines correspondences between XML attributes and stores them in an SRML format. It is not necessary to store attributes which are computable with the SRML description - this is the reduced version of the XML document.

[4]These are newly defined attributes which differ from those in XML files.

The complete document is recoverable from the reduced version and the SRML description.

We can use learning to create an SRML description from an XML document. It can supply us with a description of valid correspondences between attributes. We can utilize it to compress the document because the sum total of the given SRML description and the reduced version of the document is generally much smaller than that for the entire document.

We are developing a tool which will be able to handle a general SRML description and use both the *complete* and *reduce* methods.

The theory of learning SRML descriptions is an interesting and open area of research. In future experiments we plan to extend our tool with various learning modules so as learn SRML descriptions from any XML documents, keeping on eye on the size of the reduced version of documents.

References

[1] H. Alblas, *Introduction to attribute grammars*, Springer Verlag, In Proc. of SAGA (H. Alblas and B.Melichar eds.) LNCS **545** (1991), 1–16.

[2] T. Bray, J. Paoli, and C. Sperberg-MacQueen, *Extendable markup language*, 1998.

[3] R. Ferenc, *CPPML - An Implementation of the Columbus Schema for C++*.

[4] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen, *Tool for reverse engineering large object oriented software*, SPLST (2001), 16–27.

[5] T. Gyimóthy and T. Horváth, *Learning semantic functions of attribute grammars*, Nordic Journal of Computing **4** (1997), no. 3, 287–302.

[6] U. Kastens, *Oredered attribute grammars*, Acta Informatica **13** (1980), 229–256.

[7] D. E. Knuth, *Semantics of context-free languages*, vol. 2, pp. 127–145, Springer-Verlag, New York, 1968.

[8] T. Mitchell, *Machine learning*, McGraw-Hill, 1997.

[9] G. Psaila and S. Crespi-Reghizzi, *Adding Semantics to XML*, Second Workshop on Attribute Grammars and their Applications, WAGA'99 (Amsterdam, The Netherlands) (D. Parigot and M. Mernik, eds.), INRIA rocquencourt, 1999, pp. 113–132.

[10] XML Compression Tools, *http://sourceforge.net/projects/xmlppm/*.

[11] XML parsers, *XML-sofware. http://www.xmlsoftware.com/parsers.html.*

[12] Sz. Zvada and T. Gyimóthy, *Using decision trees to infer semantics functions of attribute grammars*, Acta Cybernetica **15** (2001), 279–304.