# Compacting XML Documents

Miklós Kálmán, Ferenc Havasi, and Tibor Gyimóthy *

University of Szeged

**Abstract.** Nowadays one of the most common formats for storing information is XML. The size of XML documents can be rather large, and they may contain redundant attributes which can be calculated from others. The main idea behind our paper is based on a relationship between XML documents and attribute grammars. Using this relationship it is possible to define semantic rules for XML attributes using a metalanguage called SRML. With this metalanguage we decided to develop a method for compacting XML documents. After compaction it is possible to use XML compressors to make the compacted document smaller, thus increasing the potential compression ratio of the compressors. Devising the rules can be done manually or by a machine learning approach. Our method can be viewed as a form of data mining, meaning that it can find relationships between attributes which might not have been noticed by the user beforehand.

## 1 Introduction

These days it seems that XML documents are becoming ever more important. The number of applications capable of storing things in XML format is growing quite rapidly. If the growth continues at this rate, XML documents will span every area in computing.

XML documents can be quite large, but many systems can only handle smaller files (e.g: embedded systems). Size is also important when an XML document has to be transferred via a network. One solution is to compress the documents using a general (e.g. zip) or XML compressor (XMill [11]). Unfortunately the compressed size of the files may still be too large.

The XML document may of course contain dependencies which are not discoverable by the above-mentioned compressors. One of these dependencies could be a relationship between two attributes, where it might be possible to calculate one from the other. Our method offers a solution to this problem, employing a special (SRML: Semantic Rule Meta Language) file format for storing the rules. These SRML rules describe how the value of an attribute can be calculated from the values of other attributes. These rules are quite similar to those of the semantic functions of Attribute Grammars, and can be used to compact the XML document by removing computable attributes.

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

The generation of these SRML files can be done manually (if the relationship between attributes is known) or via machine learning methods. The method examines the relationship between the attributes and looks for patterns in them using specific rules. We implemented our algorithm in JAVA in order to make the modules more portable and platform independent. The whole implementation is based on a framework system (every algorithm is considered as a plug-in).

During the testing of the implementation, the input XML files were compacted to 70-80% of their original size, without loss of compressibility (e.g. the XMill XML compressor could compress the compacted file with about the same efficiency as the original document. Needless to say, the file which was compressed after first being compacted was much smaller). The increased compressibility of XML files is the main advantage of our method, apart from gaining a general understanding of the relationships between attributes. To test our new method we used an XML exchange format called CPPML [4]. (CPPML is a metalanguage which describes C++ programs) When a CPPML document was compacted via this technique, followed by XMill, the compressibility ratio increased by 10% (of the original compressed size). This could make the new method an useful partner in future XML compressors. The method operates using manually generated rules, but the effectiveness of an SRML file created via machine learning can attain that of manual SRML generation.

In this article we examine the main idea previously published in [6], and afterwards present a new result, that of rule generating via machine learning methods.

In the following sections some background knowledge will first be provided. Then an overview of our method will be given using examples to illustrate how it works. Afterwards the modules of the implementation will be thoroughly described along with their detailed function. Following this, a section will explain how the learning of SRML files is achieved and the advantages these rules offer. Finally, we round off the paper by mentioning related works and a brief summary.

## 2 Preliminaries

In this section a basic introduction to XML files will be given along with the necessary preliminaries for formal and attribute grammars. This will be needed to better understand parts in the subsequent sections.

### 2.1 XML

XML documents are very similar to *html* files, as they are both text-based. The components in both are called *elements*, which might contain further *elements* and/or text, or they may be empty. *Elements* may have attributes like the attribute *href* of html tag *a*. In Figure 1 there is an example for storing a numeric expression in XML format.

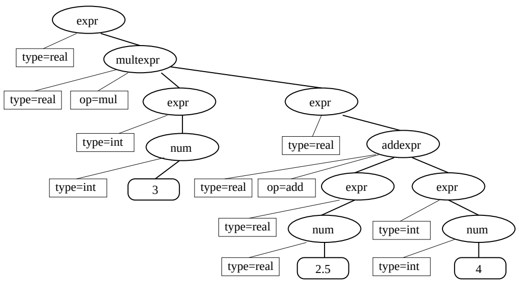
```
<expr>
  <multexpr op="mul" type="real">
    <expr type="int"> <num type="int">3</num> </expr>
    <expr type="real">
      <addexpr op="add" type="real">
        <expr type="real"> <num type="real">2.5</num> </expr>
        <expr type="int"> <num type="int">4</num> </expr>
       </addexpr>
   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

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

### 2.2 Formal languages and Attribute grammars

After defining the XML structure, we will introduce some background knowledge on the grammars of formal languages. Defining a basic syntax can be done through a formal grammar. A formal grammar [1] is a 4-tuple G=(N,T,S,P), where N is the set of nonterminal symbols, T is a set of terminal symbols, S a start-symbol and P a set of transformation rules. Given a grammar, a derivation tree can be generated based on a specific input. Next the Attribute Grammars have to be introduced since they are quite similar to SRML rules. An Attribute Grammar [7] contains a context free grammar, attributes and semantic rules. With the help of attribute grammars, attributed derivation trees can be created according to their attributes based on a specific input.

### 2.3 The relationship between XML and Attribute Grammars



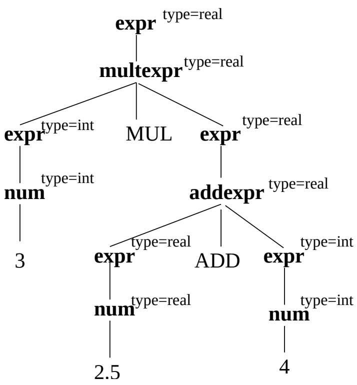




**Fig. 2.** (a) XML document DOM tree, (b) Attributed Derivation Tree

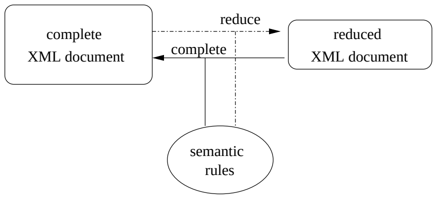
Examining the attributed derivation tree and the DOM tree in *Figure 2*, it is clear that the XML document can be considered as an attributed derivation tree as well. Actually, the following analogy exists between AG and XML documents. In Attribute Grammars, the Nonterminals correspond to the elements in the XML document. Syntactic Rules are presented as an element type declaration in the DTD of the XML file (DTD can define the structure of similar XML documents like books or CPPML descriptions). An attribute specification in the AG corresponds to an attribute list declaration in the DTD. In addition to this there is an important concept in Attribute Grammars which has no XML counterpart; the semantic functions. It might be useful to apply these semantic functions in the XML environment as well.

The attribute instances and their values are stored directly in the XML document files. If it were possible to define semantic functions, it would be enough to store the rules applying to specific attributes, since their correct values could then be calculated. With the help of this it would then be possible to avoid having to store those attributes which could be calculated. Our idea was to define the XML attributes with the aid of semantic functions. The definition of semantic rules would be an integral part of XML document files.

## 3 An approach for the compaction of XML documents

### 3.1 Compacting / Decompacting

After mentioning the general similarities concerning AG and XML, a simple example will be given to show how attributes of XML documents can be computed.




In the figure above it is clear that the *decompacting (complete)* procedure expands the document and recreates the original XML document. The process uses the semantic rules to calculate those attribute values which were previously provided. Every attribute which was not defined in the compacted XML document will be restored if there is a rule in the semantics rules set for it.

The *compacting (reduce)* procedure does just the opposite. The input used is a complete XML document with an attached semantic rules file. Every attribute which can be correctly calculated using the attached rules will be removed. This results in a reduced, compacted XML document. If the semantic rule for a given attribute does not give the correct value the attribute is not removed, maintaining the document's integrity.

After the XML document has been compacted it can then be compressed using gzip or XMill (XML compressor) without loss of compression efficiency.

### 3.2 The SRML Meta language

When we apply the compacting/decompacting procedures, the semantic functions have to be stored. An XML-based metalanguage called *SRML (Semantic Rule Meta Language)* has been defined to describe semantic rules [6]. The DTD of SRML files is as follows:

```
<!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)>
<!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 position EMPTY>
<!ATTLIST position  element NMTOKEN "srml:all"
                    from (begin | current | end) "begin">
<!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 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)>
```

This DTD can be regarded as a formal grammar: the elements become the nonterminals and the element definitions are the formal rules. The following part will describe the meaning of each SRML element defined in the DTD :

**semantic-rules:** This is the root element of the SRML file

**rules-for:** This element collects the semantic rules of a transformation rule. In the DTD there is only one definition for each element, hence a transformation rule can be defined via this element (this takes the left side of the transformation rule). This is the root attribute of the *rules-for* expression.

**rule:** This element defines the semantic rule. It must be say which *attribute* of which *element* the semantic rule is being defined and provide the value in the *expr*. If the value of *element* is *srml:root* then an attribute of the context's root is being defined.

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

**if-element:** The definition of a DTD element can contain regular expressions (using the +,*,? symbols). This element allows for the testing of the inputs form. It contains two expr elements. Like all conditional expressions the value of the *if-element* can be true or false depending on the following: if the name of the first *expr*th child (element) is equal to the second *expr* value then the return value is true, otherwise it is false. The *from* attribute defines the starting point of the examination. This also allows us to examine the last child without knowing the actual number of siblings.

**binary-op:** This element is a simple binary expression.

**position:** Returns a 0-based index that defines the current attribute's position relative to its siblings, taking into account the *element* attribute. The possible directions are *begin* and *end*. It is possible to use the *srml:all* constant as well, which results in an index describing which child the element is on the DOM level. If an *element* name is provided, then the returned index will be $n$, where the examined element has exactly $n$ predecessors or successors with the same name (depending on the direction traversed).

**attribute:** The *attribute* is defined by the *element, attrib, from* and *num* attributes. In the the environment this is the *num*th element with the name of *element's* value (if this is *srml:any* then it can be anything, if it is *srml:root* then an attribute of the root element is being referred to), the direction (*from*) can be *begin* , *end* , *current* (from the current index). If no such attribute exists the return value will be *no-data*.

**if-expr:** This is a traditional conditional expression. The return value will be based on the value of the first *expr* expression. If the first expression is evaluated then the return value will be the that of the second expression, otherwise the value will be the that of the third expression.

**if-all:** This is an iterated version of the previous *if-expr*. The value of the first *expr* is calculated with the values of the matching attributes (everything that matches the element and attribute mask, which can be a given value or *srml:all*). To refer to the value of the current attribute the *current-attribute* element should be used. If the first condition is true (first *expr*) for all matching attributes then the value will be the that of the second *expr*, otherwise it will be that of the third expression.

**if-any:** This is similar to *if-all*, but here it is enough that at least one attribute matches the condition.

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

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

**no-data:** This element says that the value of this attribute cannot be defined. It is usual to apply this in specific branches of conditional expressions.

**extern-function:** This element calls an external function which depends on the implementation. This makes the SRML more extendable.

**param:** Defines the parameter of the extern-function.

A correct SRML definition has to be consistent. This means that an attribute instance can only have one corresponding rule. Examples of this can be found on page 12.

## 4 SRMLTool: a compactor for XML documents

We have implemented a tool for the compaction of XML documents. This tool is based on semantic rules written in SRML, describing the attributes of the XML documents. It means that the inputs of the tool are an XML document and an SRML file to define semantic (computation) rules for some attributes of the document. The output is a reduced document (some attributes are removed). Naturally the tool is able to reconstruct the original XML file from the reduced document. *Figure 3(a)* shows the modular structure of the package.

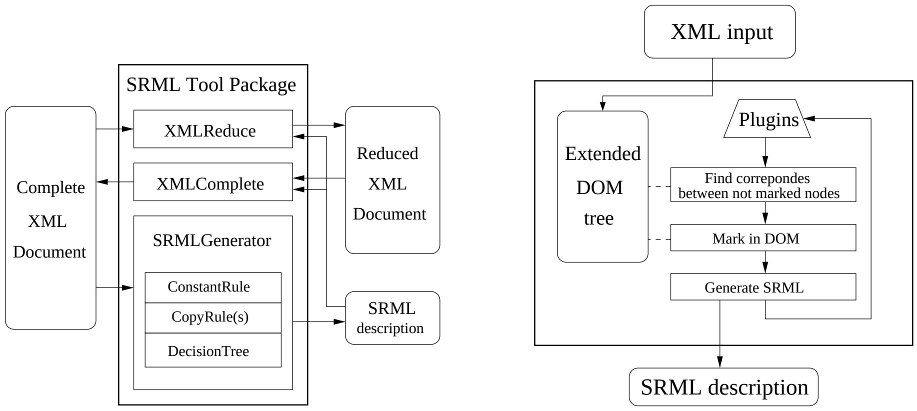



**Fig. 3.** (a) SRMLTool package, (b) Learning SRML Rules

### 4.1 The Reduce algorithm (compacting)

During the implementation of the *Reduce* (compacting) algorithm, the function of the *Complete* (decompacting) algorithm had to be considered. If there is a rule for a non-defined attribute it has to be marked somehow. If this is ignored, the *Complete* algorithm will insert a value for it and make the *compaction/decompaction* process inconsistent. To remedy this problem the *srml:var* attribute is introduced into the *compacted* document. This attribute marks the name of those attributes which were not present in the original document. Below is a simple example for this:

```
<book name="mybook">
  <section author="authorID" footNoteID="1"/>
  <section author="authorID" footNodeID="1"/>
  <section/>
</book>
```

Supposing there are rules for *section.author* and *section.footNoteID* the output would be the following (the third section does not have these attributes so they must be marked):

```
<book name="mybook">
  <section/>
  <section/>
  <section srml:var=" author footNoteID " />
</book>
```

There is another interesting point of the implementation: there may be loops in the dependencies. For example if the rules $A.x = B.x$, $B.x = C.x$ and $C.x = A.x$ are given there is an exact rule for every attribute, but only two of them can be deleted if it needs to be restored later. To resolve dependency issues the following algorithm is used:

1. Create a dependency list (every attribute has both input and output dependencies, the input dependencies being those attributes upon which it depends and the output are those that depend on it. The dependencies are represented as vertices)
2. Look for an attribute that has no input or output vertex. If there is one then it can be deleted.
3. Look for an attribute which has no output vertex.
4. If there is one, delete the attribute and all its input vertices.
5. If there isn't one like this, try to find one which has only output vertices (i.e. it doesn't depend on other attributes).
6. If there is one (that has only output vertices), keep this attribute and delete the vertices (keeping means that its added to a vector)
7. If there isn't one like this (circular reference), choose the first attribute, keep it and delete its vertices.
8. If the dependency list is empty then END, else Goto 2

The algorithm always terminates since the dependency list is always emptied. The vector *keep* contains those attributes which will not be removed. Note: This algorithm is not the most optimal solution, but with it reduction is safe and the completion process can be performed without any losses.

### 4.2 The Complete algorithm (decompacting)

The implementation of the algorithm starts off by reading both the XML file and SRML file into separate DOM trees. This saves a lot of time on operations performed later. Then the XML DOM tree is processed using an inorder tree visit routine that examines every node and every attribute. The purpose of this examination is to find out which attributes have corresponding SRML rules. If an attribute having an SRML rule is found it is stored in a vector (*processVec*), which is later processed. The *processVec* vector is used for decompacting, which is a two-stage operation. First a vector is created with those attributes having corresponding rules, then in stage two the vector elements are processed. This speeds up the decompacting since the DOM tree is visited only once. Afterwards tree pointers are used to access the nodes.

## 5 Learning SRML rules

In some cases the user does not know the relationship between the attributes of an XML document so he cannot therefore provide SRML rules. To overcome this problem the *SRMLGenerator* module was created, a module which plays an active part in the *SRMLTool* package. The *SRMLGenerator* module is based on a framework system so that it can be expanded later with plug-in algorithms. Every plug-in algorithm must fit the interface defined by the *SRMLInterface*. The process of learning is as follows: the program reads all the plug-in algorithms and executes them sequentially. After one of them has exited it marks those attributes for which it could find an SRML rule to in the DOM tree, making the next algorithm only process the attributes which have no rules. *Figure 3(b)* shows the process of learning.

Before describing the learning algorithms which have been implemented so far, the purpose and advantage of learning SRML rules will be presented.

SRML files have other crucial uses apart from making the XML files more compact. One of these is that SRML enables the user to discover relationships and dependencies between attributes, dependencies which might not have been seen by the user previously. In this case of course the SRML file has to be created dynamically using machine learning and other forms of learning. The SRML files created by machine learning can be an input to other systems such as decision-making systems, where the relationship between specific criteria is examined. It may be employed in Data Mining and other fields where relationships in large amounts of data are sought. The SRMLGenerator module in a sense "understands" the XML file and saves this knowledge in the SRML file.

### 5.1 The SRMLGenerator module's learning algorithms

The *SRMLGenerator* currently contains five plug-in algorithms. These algorithms can be expanded with additional plug-in algorithms thanks to our framework system. A new plug-in algorithm can be created simply by creating a class which conforms to the appropriate interface.

**SRMLConstantRule** This is the simplest learning algorithm in the package. This algorithm uses statistical analysis to retrieve the number of attribute instance values and then decides whether to make a rule for it. For instance this algorithm searches for $A.x = 4$ and $A.B.x = 4$ type of rules. The difference between these two types is that the first is synthesized while the second is inherited (see *Figure 4(a)*). The decision is based on whether the size of the new rule would

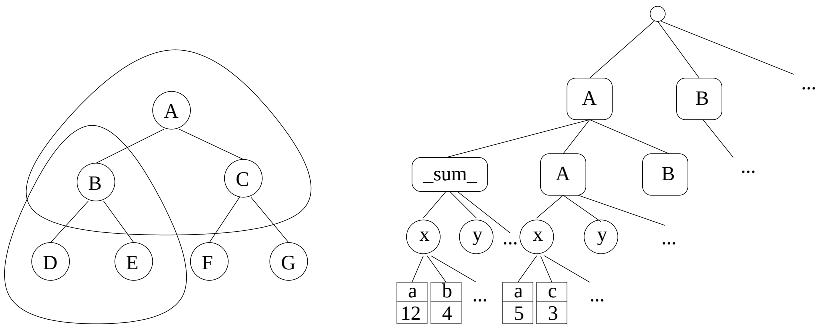



**Fig. 4.** (a) The two contexts, (b) The statistical tree of SRMLConstantRule

be bigger than that of the size decrease achieved by removing the attributes. The tree in *Figure 4(b)* is used in evaluations performed by the algorithm.

To get a clearer understanding of the tree a brief explanation will be provided of how it is built. First the input XML file is parsed and each attribute occurrence is examined. All occurrences have two counters incremented in the tree: $/elementName/\_sum\_/attribName/value$ (synthesized case) and $/parentElementName/elementName/attribName/value$ (inherited case).

After this stage the exact benefit of generating SRML rules in synthesized or inherited form can be calculated using the created statistical tree. The better one will be chosen (if a rule can be generated).

**Copy Rules** These algorithms search for $A.x = B.x$ rules. The time and memory requirements of searching for this type of rule in one stage are very high. That is why the implementation was separated into three modules: *SRMLCopyChildRule* $(x = B.x)$ *SRMLCopyAttribRule* $(x = y)$ and *SRMLCopyParentRule* $(B.y = x)$. The implementation uses similar statistical trees like above.

**SRMLDecisionTree** The *SRMLDecisionTree* plug-in is by far the most advanced in the currently implemented algorithms. It makes use of machine learning [8] in order to discover relationships and builds if-else decisions using a binary tree similar to ID3. At the start of the algorithm it creates a statistical DOM tree to select those attributes that should be further examined. This is achieved by using a heuristic dominance[1] function. Those attributes which are more dominant (compared to other attributes) are saved with both rule contexts. These contexts are saved to files, each containing one attribute and all its occurrence contexts. The next step is now to convert these files into a learning table form. This is needed because converting it not only cuts down the processing time, but also allows one to use external learning modules which use learning tables as inputs. Every attribute which needs to be learned has a separate file and will be processed later. The table form is used because, by doing this, external machine learners can also be used to find new relationships. In the implementation a decision tree similar to ID3 is used. The only difference in our case is that the tree depth is limited to 3 (this is limited because the speed of the current learning algorithm would take too long and statistics show that in most cases a depth of 3 is enough to achieve acceptable results) and a dominance function is employed.

## 6 Experimental results

The testing of our implementation was done via CPPML [4], an XML exchange format that is used as an output of Columbus Reverse Engineering package [10]. Our method can be applied to non-CPPML XML files as well as long as the elements contain attributes, like an e-book with chapters and word counts as attributes or an Oracle database dump.

### 6.1 A real sized case study: CPPML

Before showing the results of our method the CPPML [4] language should be discussed. All the input files were CPPML files. CPPML files can be created from a CPP file. CPPML is a meta language capable of describing the structure of programs written in C++. Creating CPPML files can be done via the Columbus Reverse engineering package (CPPML is XML based). [10]

To illustrate how CPPML works let us consider the following C++ program:

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

[1] This heuristic dominance function was written by us and can be changed and optimized. Currently an attribute is dominant if the attribute count is larger than the average count. This means that if a value of an attribute occurs more times than any other value then it becomes dominant, since it might be worth splitting the tree into two, based on this value. One part would be where the value of this attribute equals the dominant value and the other where it does not, making an if-else statement.

The CPPML form of the program can be the following:

```
<class id="id20097" name="_guard" path="D:\tm\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:\tm\SymbolTable\Input\CANGuard.h" line="75" end-line="76" visibility="public"
      const="no" virtual="no" pure-virtual="no" kind="normal" body-line="75"
      body-end-line="76" body-path="D:\tm\SymbolTable\Input\CANGuard.h">
        <return-type>void</return-type>
        <parameter id="id20106" name="ti" path="D:\tm\SymbolTable\Input\CANGuard.h"
            line="74" end-line="74" const="yes">
        <type>type_info&amp;</type>
        </parameter>
    </function>
    ...
</class>
...
```

It is quite obvious that in the CPPML definition a lot of attributes can be calculated or estimated via other attributes. One of these is the *kind* attribute which stores the type of the function. If the function name matches that of the class name then it is a *constructor*, but if the function name starts with a $\sim$ then it is a *destructor*.

Expressed in SRML form, this might look like the following:

```
<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>
```

Only valid rules can be defined, but sometimes it is possible to estimate rules (the *reduce* procedure will only delete those attributes which match a pattern) Consider the following estimation:

1. A function declaration starts and ends on the same line
2. The implementation of a class's function is usually in the same file as the class itself

3. The parameters of a function are usually in the same file, perhaps somewhere in the same line

Expressed in SRML form these "estimated" SRML rules may look like the following:

```
<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>
```

After running the *compaction* module the following XML document is produced:

```
<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>
  ...
</class>
```

The rules described above produced a compaction ratio of 68.9%, since the original fragment was 2.180 bytes and the compressed was 1.502 bytes. This ratio can be improved further with the introduction of new SRML rules.

### 6.2 Compacting CPPML with SRML rules created by hand

The results achieved using SRML files created by hand are shown in *Figure 5*. Input files were in CPPML form. The (C) bracket indicates that the compressors were applied to the compacted version of the XML file.

| File | SymbolTable | Jikes | AppWiz |
|---|---|---|---|
| Orig | 399321 | 2233824 | 3547297 |
| gzip (ratio) | 30460 (7.62%) | 177051 (7.92%) | 244174 (6.68%) |
| XMill (ratio) | 19786 (4.95%) | 114275 (5.11%) | 145738 (4.10%) |
| Comp (ratio) | 296193 (74.10%) | 1736267 (77.70%) | 2238308 (63.10%) |
| gzip(C) (ratio) | 26308 (6.58%) | 160609 (7.18%) | 206522 (5.82%) |
| XMill(C) (ratio) | 18008 (4.50%) | 108458 (4.85%) | 134217 (3.78%) |

**Fig. 5.** Compaction chart using manual rules

### 6.3 Compacting CPPML with machine learning SRML rules

In *Figure 6* a comparison is made between the efficiency of the machine-learned and hand-generated SRML rules. In many cases SRML generated via machine

| Filename | Manual | Machine | Diff |
|---|---|---|---|
| SymbolTable (399321) | 296193, 74.10% | 313873, 78.60% | 17680, 4.42% |
| Jikes (2233824) | 1736267, 77.70% | 1821228, 81.52% | 84961, 3.80% |
| AppWiz (3547297) | 2238308, 63.10% | 2773946, 78.19% | 535656, 15.10% |

**Fig. 6.** A comparison of machine learned and hand-generated rules

learning approaches the compaction ratio of a hand-generated SRML. These results can be improved by introducing new plug-in algorithms into the SRML-Generator module. The execution order of the plug-ins matter. That is why the machine learning *(SRMLDecisionTree)* plug-in is used first since this seems to provide the optimal solution, an observation based on experiment. The reason why this could be the most optimal is that it employs a dominance function in the decision trees as well and seems to generate more rules.

### 6.4 Resource requirements of the tool

For testing, a Debian Linux environment was used on a PC (AMD Athlon XP 1600+ 512M DDR). The package requires about 200MB of memory since the DOM tree takes up a lot of space. The AppWiz file was *compacted* in approximately 2 minutes and decompacted in 30 seconds. The execution time was long in the case of machine learning (SRMLDecisionTree) since it takes some time for the program to fully understand the relationship between the attributes. Depending on the complexity and level of recursion, the execution time varied from 2 hours to 30 hours.

## 7 Related Work

The first notion of adding semantics to XML documents was introduced in the paper *Adding Semantics to XML* [9], which had its own SRD (Semantics Rule Definition) consisting of two parts: the first one describes the semantics attributes[2], 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 [9] 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 integral part of XML document files. This kind of semantics definition could provide a useful extension for XML techniques. 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.

Another article which can be viewed as a related work is the *Learning semantic functions of attribute grammars* [5] paper, which provides a way of learning attribute grammars. The learning problem of semantic functions is transformed to a propositional form and the hypothesis induced by a propositional learner is transformed back into corresponding semantic functions. This method is similar to ours as it learns and uses semantic functions based on examples, but it only works on attributes with very small domains. It searches for precise rules in contrast to our method, which can use approximated rules as well.

## 8 Summary

One of the biggest problems associated with XML documents is that they can become rather large. Our method can make their compression more effective. This method is based on a relationship between attribute grammars and XML documents. Using the SRML metalanguage a 20-30% size decrease can be attained without loss of information or compressibility. The package we have implemented is able to compact and decompact XML files using existing rules, or it can generate rules. These rules can be used for compacting a document or for "understanding" it (e.g. Data Mining).

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

## References


1. Moll R.N. Arbib M.A. Kfoury A.J., *An introduction to formal language theory*, 1988.
2. H. Alblas, *Introduction to attribute gammars*, Springer Verlag, In Proc. of SAGA (H. Alblas and B.Melichar eds.) LNCS **545** (1991), 1–16.
3. T. Bray, J. Paoli, and C. Sperberg-McQueen, *Extensible markup language*, (XML) 1.0 (W3C recommendation) (feb 1998).
4. R. Ferenc, *CPPML - an implementation of the Columbus Schema for C++*.
5. T. Gyimóthy and T. Horváth, *Learning semantic functions of attribute grammars*, Nordic Journal of Computing **4** (1997), no. 3, 287–302.
6. F. Havasi, *XML semantics extension*, Acta Cybernetica **15** (2002), no. 2, 509–528.
7. D. E. Knuth, *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), 127–145.
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. Á. Beszédes Á. Kiss M. Tarkiainen R. Ferenc F. Magyar, *Tool for the reverse engineering of large object oriented software*, SPLST (2001), 16–27.
11. XMill, *http://www.research.att.com/sw/tools/xmlill/*.